Twisted Documentation

Release 17.9.0

Twisted Matrix Labs

Sep 23, 2017

Contents

Contents:

CHAPTER 1

Installing Twisted

Installing Optional Dependencies

This document describes the optional dependencies that Twisted supports. The dependencies are python packages that Twisted's developers have found useful either for developing Twisted itself or for developing Twisted applications.

The intended audience of this document is someone who is familiar with installing optional dependencies using pip.

If you are unfamiliar with the installation of optional dependencies, the python packaging tutorial can show you how. For a deeper explanation of what optional dependencies are and how they are declared, please see the setuptools documentation.

The following optional dependencies are supported:

- **dev - packages that aid in the development of Twisted itself.**
  - pyflakes
  - twisted-dev-tools
  - python-subunit
  - Sphinx
  - TwistedChecker, only available on python2
  - pydoctor, only available on python2
- **tls - packages that are needed to work with TLS.**
  - pyOpenSSL
  - service_identity
  - idna
- **conch - packages for working with conch/SSH.**
  - pyasn1
  - cryptography

- **soap** - the SOAPpy package to work with SOAP.
- **serial** - the pyserial package to work with serial data.
- **all_non_platform** - installs **tls**, **conch**, **soap**, and **serial** options.
- **osx_platform** - **all_non_platform** options and pyobjc to work with Objective-C apis.
- **windows_platform** - **all_non_platform** options and pypiwin32 to work with Windows's apis.
- **http2 - packages needed for http2 support.**
  - h2
  - priority
- *Installing Optional Dependencies*: documentation on how to install Twisted's optional dependencies.

CHAPTER 2

Twisted Core

Developer Guides

The Vision For Twisted

Many other documents in this repository are dedicated to defining what Twisted is. Here, I will attempt to explain not what Twisted is, but what it should be, once I've met my goals with it.

First, Twisted should be fun. It began as a game, it is being used commercially in games, and it will be, I hope, an interactive and entertaining experience for the end-user.

Twisted is a platform for developing internet applications. While Python by itself is a very powerful language, there are many facilities it lacks which other languages have spent great attention to adding. It can do this now; Twisted is a good (if somewhat idiosyncratic) pure-python framework or library, depending on how you treat it, and it continues to improve.

As a platform, Twisted should be focused on integration. Ideally, all functionality will be accessible through all protocols. Failing that, all functionality should be configurable through at least one protocol, with a seamless and consistent user-interface. The next phase of development will be focusing strongly on a configuration system which will unify many disparate pieces of the current infrastructure, and allow them to be tacked together by a non-programmer.

Writing Servers

Overview

This document explains how you can use Twisted to implement network protocol parsing and handling for TCP servers (the same code can be reused for SSL and Unix socket servers). There is a *separate document* covering UDP.

Your protocol handling class will usually subclass twisted.internet.protocol.Protocol. Most protocol handlers inherit either from this class or from one of its convenience children. An instance of the protocol class is instantiated per-connection, on demand, and will go away when the connection is finished. This means that persistent configuration is not saved in the `Protocol`.

The persistent configuration is kept in a `Factory` class, which usually inherits from twisted.internet.protocol.Factory. The `buildProtocol` method of the `Factory` is used to create a `Protocol` for each new connection.

It is usually useful to be able to offer the same service on multiple ports or network addresses. This is why the `Factory` does not listen to connections, and in fact does not know anything about the network. See *the endpoints documentation* for more information, or IReactorTCP.listenTCP and the other `IReactor*.listen*` APIs for the lower level APIs that endpoints are based on.

This document will explain each step of the way.

Protocols

As mentioned above, this, along with auxiliary classes and functions, is where most of the code is. A Twisted protocol handles data in an asynchronous manner. The protocol responds to events as they arrive from the network and the events arrive as calls to methods on the protocol.

Here is a simple example:

```
from twisted.internet.protocol import Protocol

class Echo(Protocol):

    def dataReceived(self, data):
        self.transport.write(data)
```

This is one of the simplest protocols. It simply writes back whatever is written to it, and does not respond to all events. Here is an example of a Protocol responding to another event:

```
from twisted.internet.protocol import Protocol

class QOTD(Protocol):

    def connectionMade(self):
        self.transport.write("An apple a day keeps the doctor away\r\n")
        self.transport.loseConnection()
```

This protocol responds to the initial connection with a well known quote, and then terminates the connection.

The `connectionMade` event is usually where setup of the connection object happens, as well as any initial greetings (as in the QOTD protocol above, which is actually based on **RFC 865**). The `connectionLost` event is where tearing down of any connection-specific objects is done. Here is an example:

```
from twisted.internet.protocol import Protocol

class Echo(Protocol):

    def __init__(self, factory):
        self.factory = factory

    def connectionMade(self):
        self.factory.numProtocols = self.factory.numProtocols + 1
        self.transport.write(
            "Welcome! There are currently %d open connections.\n" %
            (self.factory.numProtocols,))

    def connectionLost(self, reason):
        self.factory.numProtocols = self.factory.numProtocols - 1
```

```
    def dataReceived(self, data):
        self.transport.write(data)
```

Here `connectionMade` and `connectionLost` cooperate to keep a count of the active protocols in a shared object, the factory. The factory must be passed to `Echo.__init__` when creating a new instance. The factory is used to share state that exists beyond the lifetime of any given connection. You will see why this object is called a "factory" in the next section.

loseConnection() and abortConnection()

In the code above, `loseConnection` is called immediately after writing to the transport. The `loseConnection` call will close the connection only when all the data has been written by Twisted out to the operating system, so it is safe to use in this case without worrying about transport writes being lost. If a *producer* is being used with the transport, `loseConnection` will only close the connection once the producer is unregistered.

In some cases, waiting until all the data is written out is not what we want. Due to network failures, or bugs or maliciousness in the other side of the connection, data written to the transport may not be deliverable, and so even though `loseConnection` was called the connection will not be lost. In these cases, `abortConnection` can be used: it closes the connection immediately, regardless of buffered data that is still unwritten in the transport, or producers that are still registered. Note that `abortConnection` is only available in Twisted 11.1 and newer.

Using the Protocol

In this section, you will learn how to run a server which uses your `Protocol`.

Here is code that will run the QOTD server discussed earlier:

```
from twisted.internet.protocol import Factory
from twisted.internet.endpoints import TCP4ServerEndpoint
from twisted.internet import reactor

class QOTDFactory(Factory):
    def buildProtocol(self, addr):
        return QOTD()

# 8007 is the port you want to run under. Choose something >1024
endpoint = TCP4ServerEndpoint(reactor, 8007)
endpoint.listen(QOTDFactory())
reactor.run()
```

In this example, I create a protocol `Factory`. I want to tell this factory that its job is to build QOTD protocol instances, so I set its `buildProtocol` method to return instances of the QOTD class. Then, I want to listen on a TCP port, so I make a TCP4ServerEndpoint to identify the port that I want to bind to, and then pass the factory I just created to its `listen` method.

`endpoint.listen()` tells the reactor to handle connections to the endpoint's address using a particular protocol, but the reactor needs to be *running* in order for it to do anything. `reactor.run()` starts the reactor and then waits forever for connections to arrive on the port you've specified. You can stop the reactor by hitting Control-C in a terminal or calling `reactor.stop()`.

For more information on different ways you can listen for incoming connections, see *the documentation for the endpoints API*. For more information on using the reactor, see *the reactor overview*.

Helper Protocols

Many protocols build upon similar lower-level abstractions.

For example, many popular internet protocols are line-based, containing text data terminated by line breaks (commonly CR-LF), rather than containing straight raw data. However, quite a few protocols are mixed - they have line-based sections and then raw data sections. Examples include HTTP/1.1 and the Freenet protocol.

For those cases, there is the LineReceiver protocol. This protocol dispatches to two different event handlers – `lineReceived` and `rawDataReceived`. By default, only `lineReceived` will be called, once for each line. However, if `setRawMode` is called, the protocol will call `rawDataReceived` until `setLineMode` is called, which returns it to using `lineReceived`. It also provides a method, `sendLine`, that writes data to the transport along with the delimiter the class uses to split lines (by default, `\r\n`).

Here is an example for a simple use of the line receiver:

```
from twisted.protocols.basic import LineReceiver

class Answer(LineReceiver):

    answers = {'How are you?': 'Fine', None: "I don't know what you mean"}

    def lineReceived(self, line):
        if line in self.answers:
            self.sendLine(self.answers[line])
        else:
            self.sendLine(self.answers[None])
```

Note that the delimiter is not part of the line.

Several other helpers exist, such as a netstring based protocol and prefixed-message-length protocols.

State Machines

Many Twisted protocol handlers need to write a state machine to record the state they are at. Here are some pieces of advice which help to write state machines:

- Don't write big state machines. Prefer to write a state machine which deals with one level of abstraction at a time.
- Don't mix application-specific code with Protocol handling code. When the protocol handler has to make an application-specific call, keep it as a method call.

Factories

Simpler Protocol Creation

For a factory which simply instantiates instances of a specific protocol class, there is a simpler way to implement the factory. The default implementation of the `buildProtocol` method calls the `protocol` attribute of the factory to create a `Protocol` instance, and then sets an attribute on it called `factory` which points to the factory itself. This lets every `Protocol` access, and possibly modify, the persistent configuration. Here is an example that uses these features instead of overriding `buildProtocol`:

```
from twisted.internet.protocol import Factory, Protocol
from twisted.internet.endpoints import TCP4ServerEndpoint
from twisted.internet import reactor
```

```
class QOTD(Protocol):

    def connectionMade(self):
        # self.factory was set by the factory's default buildProtocol:
        self.transport.write(self.factory.quote + '\r\n')
        self.transport.loseConnection()


class QOTDFactory(Factory):

    # This will be used by the default buildProtocol to create new protocols:
    protocol = QOTD

    def __init__(self, quote=None):
        self.quote = quote or 'An apple a day keeps the doctor away'

endpoint = TCP4ServerEndpoint(reactor, 8007)
endpoint.listen(QOTDFactory("configurable quote"))
reactor.run()
```

If all you need is a simple factory that builds a protocol without any additional behavior, Twisted 13.1 added Factory.forProtocol, an even simpler approach.

Factory Startup and Shutdown

A Factory has two methods to perform application-specific building up and tearing down (since a Factory is frequently persisted, it is often not appropriate to do them in `__init__` or `__del__`, and would frequently be too early or too late).

Here is an example of a factory which allows its Protocols to write to a special log-file:

```
from twisted.internet.protocol import Factory
from twisted.protocols.basic import LineReceiver


class LoggingProtocol(LineReceiver):

    def lineReceived(self, line):
        self.factory.fp.write(line + '\n')


class LogfileFactory(Factory):

    protocol = LoggingProtocol

    def __init__(self, fileName):
        self.file = fileName

    def startFactory(self):
        self.fp = open(self.file, 'a')

    def stopFactory(self):
        self.fp.close()
```

Putting it All Together

As a final example, here's a simple chat server that allows users to choose a username and then communicate with other users. It demonstrates the use of shared state in the factory, a state machine for each individual protocol, and communication between different protocols.

chat.py

```
from twisted.internet.protocol import Factory
from twisted.protocols.basic import LineReceiver
from twisted.internet import reactor

class Chat(LineReceiver):

    def __init__(self, users):
        self.users = users
        self.name = None
        self.state = "GETNAME"

    def connectionMade(self):
        self.sendLine("What's your name?")

    def connectionLost(self, reason):
        if self.name in self.users:
            del self.users[self.name]

    def lineReceived(self, line):
        if self.state == "GETNAME":
            self.handle_GETNAME(line)
        else:
            self.handle_CHAT(line)

    def handle_GETNAME(self, name):
        if name in self.users:
            self.sendLine("Name taken, please choose another.")
            return
        self.sendLine("Welcome, %s!" % (name,))
        self.name = name
        self.users[name] = self
        self.state = "CHAT"

    def handle_CHAT(self, message):
        message = "<%s> %s" % (self.name, message)
        for name, protocol in self.users.iteritems():
            if protocol != self:
                protocol.sendLine(message)


class ChatFactory(Factory):

    def __init__(self):
        self.users = {} # maps user names to Chat instances

    def buildProtocol(self, addr):
        return Chat(self.users)


reactor.listenTCP(8123, ChatFactory())
reactor.run()
```

The only API you might not be familiar with is `listenTCP`. listenTCP is the method which connects a `Factory` to the network. This is the lower-level API that *endpoints* wraps for you.

Here's a sample transcript of a chat session (emphasised text is entered by the user):

```
$ telnet 127.0.0.1 8123
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
What's your name?
test
Name taken, please choose another.
bob
Welcome, bob!
hello
<alice> hi bob
twisted makes writing servers so easy!
<alice> I couldn't agree more
<carrol> yeah, it's great
```

Writing Clients

Overview

Twisted is a framework designed to be very flexible, and let you write powerful clients. The cost of this flexibility is a few layers in the way to writing your client. This document covers creating clients that can be used for TCP, SSL and Unix sockets. UDP is covered *in a different document* .

At the base, the place where you actually implement the protocol parsing and handling, is the `Protocol` class. This class will usually be descended from twisted.internet.protocol.Protocol . Most protocol handlers inherit either from this class or from one of its convenience children. An instance of the protocol class will be instantiated when you connect to the server and will go away when the connection is finished. This means that persistent configuration is not saved in the `Protocol` .

The persistent configuration is kept in a `Factory` class, which usually inherits from twisted.internet.protocol.Factory (or twisted.internet.protocol.ClientFactory : see below). The default factory class just instantiates the `Protocol` and then sets the protocol's `factory` attribute to point to itself (the factory). This lets the `Protocol` access, and possibly modify, the persistent configuration.

Protocol

As mentioned above, this and auxiliary classes and functions are where most of the code is. A Twisted protocol handles data in an asynchronous manner. This means that the protocol never waits for an event, but rather responds to events as they arrive from the network.

Here is a simple example:

```
from twisted.internet.protocol import Protocol
from sys import stdout

class Echo(Protocol):
    def dataReceived(self, data):
        stdout.write(data)
```

This is one of the simplest protocols. It just writes whatever it reads from the connection to standard output. There are many events it does not respond to. Here is an example of a `Protocol` responding to another event:

```
from twisted.internet.protocol import Protocol

class WelcomeMessage(Protocol):
    def connectionMade(self):
        self.transport.write("Hello server, I am the client!\r\n")
        self.transport.loseConnection()
```

This protocol connects to the server, sends it a welcome message, and then terminates the connection.

The connectionMade event is usually where set up of the `Protocol` object happens, as well as any initial greetings (as in the `WelcomeMessage` protocol above). Any tearing down of `Protocol` -specific objects is done in connectionLost .

Simple, single-use clients

In many cases, the protocol only needs to connect to the server once, and the code just wants to get a connected instance of the protocol. In those cases twisted.internet.endpoints provides the appropriate API, and in particular connectProtocol which takes a protocol instance rather than a factory.

```
from twisted.internet import reactor
from twisted.internet.protocol import Protocol
from twisted.internet.endpoints import TCP4ClientEndpoint, connectProtocol

class Greeter(Protocol):
    def sendMessage(self, msg):
        self.transport.write("MESSAGE %s\n" % msg)

def gotProtocol(p):
    p.sendMessage("Hello")
    reactor.callLater(1, p.sendMessage, "This is sent in a second")
    reactor.callLater(2, p.transport.loseConnection)

point = TCP4ClientEndpoint(reactor, "localhost", 1234)
d = connectProtocol(point, Greeter())
d.addCallback(gotProtocol)
reactor.run()
```

Regardless of the type of client endpoint, the way to set up a new connection is simply pass it to connectProtocol along with a protocol instance. This means it's easy to change the mechanism you're using to connect, without changing the rest of your program. For example, to run the greeter example over SSL, the only change required is to instantiate an SSL4ClientEndpoint instead of a `TCP4ClientEndpoint` . To take advantage of this, functions and methods which initiates a new connection should generally accept an endpoint as an argument and let the caller construct it, rather than taking arguments like 'host' and 'port' and constructing its own.

For more information on different ways you can make outgoing connections to different types of endpoints, as well as parsing strings into endpoints, see *the documentation for the endpoints API* .

You may come across code using ClientCreator , an older API which is not as flexible as the endpoint API. Rather than calling `connect` on an endpoint, such code will look like this:

```
from twisted.internet.protocol import ClientCreator

...
```

```
creator = ClientCreator(reactor, Greeter)
d = creator.connectTCP("localhost", 1234)
d.addCallback(gotProtocol)
reactor.run()
```

In general, the endpoint API should be preferred in new code, as it lets the caller select the method of connecting.

ClientFactory

Still, there's plenty of code out there that uses lower-level APIs, and a few features (such as automatic reconnection) have not been re-implemented with endpoints yet, so in some cases they may be more convenient to use.

To use the lower-level connection APIs, you will need to call one of the *reactor.connect** methods directly. For these cases, you need a ClientFactory . The `ClientFactory` is in charge of creating the `Protocol` and also receives events relating to the connection state. This allows it to do things like reconnect in the event of a connection error. Here is an example of a simple `ClientFactory` that uses the `Echo` protocol (above) and also prints what state the connection is in.

```
from twisted.internet.protocol import Protocol, ClientFactory
from sys import stdout

class Echo(Protocol):
    def dataReceived(self, data):
        stdout.write(data)

class EchoClientFactory(ClientFactory):
    def startedConnecting(self, connector):
        print('Started to connect.')

    def buildProtocol(self, addr):
        print('Connected.')
        return Echo()

    def clientConnectionLost(self, connector, reason):
        print('Lost connection.  Reason:', reason)

    def clientConnectionFailed(self, connector, reason):
        print('Connection failed. Reason:', reason)
```

To connect this `EchoClientFactory` to a server, you could use this code:

```
from twisted.internet import reactor
reactor.connectTCP(host, port, EchoClientFactory())
reactor.run()
```

Note that clientConnectionFailed is called when a connection could not be established, and that clientConnectionLost is called when a connection was made and then disconnected.

Reactor Client APIs

connectTCP

IReactorTCP.connectTCP provides support for IPv4 and IPv6 TCP clients. The `host` argument it accepts can be either a hostname or an IP address literal. In the case of a hostname, the reactor will automatically resolve the name to an IP address before attempting the connection. This means that for a hostname with multiple address records,

reconnection attempts may not always go to the same server (see below). It also means that there is name resolution overhead for each connection attempt. If you are creating many short-lived connections (typically around hundreds or thousands per second) then you may want to resolve the hostname to an address first and then pass the address to `connectTCP` instead.

Reconnection

Often, the connection of a client will be lost unintentionally due to network problems. One way to reconnect after a disconnection would be to call `connector.connect()` when the connection is lost:

```
from twisted.internet.protocol import ClientFactory

class EchoClientFactory(ClientFactory):
    def clientConnectionLost(self, connector, reason):
        connector.connect()
```

The connector passed as the first argument is the interface between a connection and a protocol. When the connection fails and the factory receives the `clientConnectionLost` event, the factory can call `connector.connect()` to start the connection over again from scratch.

However, most programs that want this functionality should implement ReconnectingClientFactory instead, which tries to reconnect if a connection is lost or fails and which exponentially delays repeated reconnect attempts.

Here is the `Echo` protocol implemented with a `ReconnectingClientFactory`:

```
from twisted.internet.protocol import Protocol, ReconnectingClientFactory
from sys import stdout

class Echo(Protocol):
    def dataReceived(self, data):
        stdout.write(data)

class EchoClientFactory(ReconnectingClientFactory):
    def startedConnecting(self, connector):
        print('Started to connect.')

    def buildProtocol(self, addr):
        print('Connected.')
        print('Resetting reconnection delay')
        self.resetDelay()
        return Echo()

    def clientConnectionLost(self, connector, reason):
        print('Lost connection.  Reason:', reason)
        ReconnectingClientFactory.clientConnectionLost(self, connector, reason)

    def clientConnectionFailed(self, connector, reason):
        print('Connection failed. Reason:', reason)
        ReconnectingClientFactory.clientConnectionFailed(self, connector,
                                                         reason)
```

A Higher-Level Example: ircLogBot

Overview of ircLogBot

The clients so far have been fairly simple. A more complicated example comes with Twisted Words in the `doc/words/examples` directory.

`ircLogBot.py`

```
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.


"""
An example IRC log bot - logs a channel's events to a file.

If someone says the bot's name in the channel followed by a ':',
e.g.

    <foo> logbot: hello!

the bot will reply:

    <logbot> foo: I am a log bot

Run this script with two arguments, the channel name the bot should
connect to, and file to log to, e.g.:

    $ python ircLogBot.py test test.log

will log channel #test to the file 'test.log'.

To run the script:

    $ python ircLogBot.py <channel> <file>
"""


from __future__ import print_function

# twisted imports
from twisted.words.protocols import irc
from twisted.internet import reactor, protocol
from twisted.python import log

# system imports
import time, sys


class MessageLogger:
    """
    An independent logger class (because separation of application
    and protocol logic is a good thing).
    """
    def __init__(self, file):
        self.file = file

    def log(self, message):
        """Write a message to the file."""
        timestamp = time.strftime("[%H:%M:%S]", time.localtime(time.time()))
```

```
        self.file.write('%s %s\n' % (timestamp, message))
        self.file.flush()

    def close(self):
        self.file.close()


class LogBot(irc.IRCClient):
    """A logging IRC bot."""

    nickname = "twistedbot"

    def connectionMade(self):
        irc.IRCClient.connectionMade(self)
        self.logger = MessageLogger(open(self.factory.filename, "a"))
        self.logger.log("[connected at %s]" %
                        time.asctime(time.localtime(time.time())))

    def connectionLost(self, reason):
        irc.IRCClient.connectionLost(self, reason)
        self.logger.log("[disconnected at %s]" %
                        time.asctime(time.localtime(time.time())))
        self.logger.close()


    # callbacks for events

    def signedOn(self):
        """Called when bot has successfully signed on to server."""
        self.join(self.factory.channel)

    def joined(self, channel):
        """This will get called when the bot joins the channel."""
        self.logger.log("[I have joined %s]" % channel)

    def privmsg(self, user, channel, msg):
        """This will get called when the bot receives a message."""
        user = user.split('!', 1)[0]
        self.logger.log("<%s> %s" % (user, msg))

        # Check to see if they're sending me a private message
        if channel == self.nickname:
            msg = "It isn't nice to whisper!  Play nice with the group."
            self.msg(user, msg)
            return

        # Otherwise check to see if it is a message directed at me
        if msg.startswith(self.nickname + ":"):
            msg = "%s: I am a log bot" % user
            self.msg(channel, msg)
            self.logger.log("<%s> %s" % (self.nickname, msg))

    def action(self, user, channel, msg):
        """This will get called when the bot sees someone do an action."""
        user = user.split('!', 1)[0]
        self.logger.log("* %s %s" % (user, msg))

    # irc callbacks
```

```
    def irc_NICK(self, prefix, params):
        """Called when an IRC user changes their nickname."""
        old_nick = prefix.split('!')[0]
        new_nick = params[0]
        self.logger.log("%s is now known as %s" % (old_nick, new_nick))


    # For fun, override the method that determines how a nickname is changed on
    # collisions. The default method appends an underscore.
    def alterCollidedNick(self, nickname):
        """
        Generate an altered version of a nickname that caused a collision in an
        effort to create an unused related name for subsequent registration.
        """
        return nickname + '^'



class LogBotFactory(protocol.ClientFactory):
    """A factory for LogBots.

    A new protocol instance will be created each time we connect to the server.
    """

    def __init__(self, channel, filename):
        self.channel = channel
        self.filename = filename

    def buildProtocol(self, addr):
        p = LogBot()
        p.factory = self
        return p

    def clientConnectionLost(self, connector, reason):
        """If we get disconnected, reconnect to server."""
        connector.connect()

    def clientConnectionFailed(self, connector, reason):
        print("connection failed:", reason)
        reactor.stop()


if __name__ == '__main__':
    # initialize logging
    log.startLogging(sys.stdout)

    # create factory protocol and application
    f = LogBotFactory(sys.argv[1], sys.argv[2])

    # connect factory to this host and port
    reactor.connectTCP("irc.freenode.net", 6667, f)

    # run bot
    reactor.run()
```

`ircLogBot.py` connects to an IRC server, joins a channel, and logs all traffic on it to a file. It demonstrates some of the connection-level logic of reconnecting on a lost connection, as well as storing persistent data in the `Factory`.

Persistent Data in the Factory

Since the `Protocol` instance is recreated each time the connection is made, the client needs some way to keep track of data that should be persisted. In the case of the logging bot, it needs to know which channel it is logging, and where to log it.

```
from twisted.words.protocols import irc
from twisted.internet import protocol

class LogBot(irc.IRCClient):

    def connectionMade(self):
        irc.IRCClient.connectionMade(self)
        self.logger = MessageLogger(open(self.factory.filename, "a"))
        self.logger.log("[connected at %s]" %
                        time.asctime(time.localtime(time.time())))

    def signedOn(self):
        self.join(self.factory.channel)


class LogBotFactory(protocol.ClientFactory):

    def __init__(self, channel, filename):
        self.channel = channel
        self.filename = filename

    def buildProtocol(self, addr):
        p = LogBot()
        p.factory = self
        return p
```

When the protocol is created, it gets a reference to the factory as `self.factory` . It can then access attributes of the factory in its logic. In the case of `LogBot` , it opens the file and connects to the channel stored in the factory.

Factories have a default implementation of `buildProtocol`. It does the same thing the example above does using the `protocol` attribute of the factory to create the protocol instance. In the example above, the factory could be rewritten to look like this:

```
class LogBotFactory(protocol.ClientFactory):
    protocol = LogBot

    def __init__(self, channel, filename):
        self.channel = channel
        self.filename = filename
```

Further Reading

The Protocol class used throughout this document is a base implementation of IProtocol used in most Twisted applications for convenience. To learn about the complete `IProtocol` interface, see the API documentation for IProtocol .

The `transport` attribute used in some examples in this document provides the ITCPTransport interface. To learn about the complete interface, see the API documentation for ITCPTransport .

Interface classes are a way of specifying what methods and attributes an object has and how they behave. See the *Components: Interfaces and Adapters* document for more information on using interfaces in Twisted.

Test-driven development with Twisted

Writing good code is hard, or at least it can be. A major challenge is to ensure that your code remains correct as you add new functionality.

Unit testing is a modern, light-weight testing methodology in widespread use in many programming languages. Development that relies on unit tests is often referred to as Test-Driven Development (TDD). Most Twisted code is tested using TDD.

To gain a solid understanding of unit testing in Python, you should read the unittest – Unit testing framework chapter of the Python LibraryReference. There is a lot of information available online and in books.

Introductory example of Python unit testing

This document is principally a guide to Trial, Twisted's unit testing framework. Trial is based on Python's unit testing framework. While we do not aim to give a comprehensive guide to general Python unit testing, it will be helpful to consider a simple non-networked example before expanding to cover networking code that requires the special capabilities of Trial. If you are already familiar with unit test in Python, jump straight to the section specific to *testing Twisted code*.

Note: In what follows we will make a series of refinements to some simple classes. In order to keep the examples and source code links complete and to allow you to run Trial on the intermediate results at every stage, I add `_N` (where the `N` are successive integers) to file names to keep them separate. This is a minor visual distraction that should be ignored.

Creating an API and writing tests

We'll create a library for arithmetic calculation. First, create a project structure with a directory called `calculus` containing an empty `__init__.py` file.

Then put the following simple class definition API into `calculus/base_1.py` :

`base_1.py`

```
# -*- test-case-name: calculus.test.test_base_1 -*-



class Calculation(object):
    def add(self, a, b):
        pass

    def subtract(self, a, b):
        pass

    def multiply(self, a, b):
        pass

    def divide(self, a, b):
        pass
```

(Ignore the `test-case-name` comment for now. You'll see why that's useful *below* .)

We've written the interface, but not the code. Now we'll write a set of tests. At this point of development, we'll be expecting all tests to fail. Don't worry, that's part of the point. Once we have a test framework functioning, and we

have some decent tests written (and failing!), we'll go and do the actual development of our calculation API. This is the preferred way to work for many people using TDD - write tests first, make sure they fail, then do development. Others are not so strict and write tests after doing the development.

Create a `test` directory beneath `calculus` , with an empty `__init__.py` file. In a `calculus/test/test_base_1.py`, put the following:

`test_base_1.py`

```
from calculus.base_1 import Calculation
from twisted.trial import unittest

class CalculationTestCase(unittest.TestCase):
    def test_add(self):
        calc = Calculation()
        result = calc.add(3, 8)
        self.assertEqual(result, 11)

    def test_subtract(self):
        calc = Calculation()
        result = calc.subtract(7, 3)
        self.assertEqual(result, 4)

    def test_multiply(self):
        calc = Calculation()
        result = calc.multiply(12, 5)
        self.assertEqual(result, 60)

    def test_divide(self):
        calc = Calculation()
        result = calc.divide(12, 5)
        self.assertEqual(result, 2)
```

You should now have the following 4 files:

```
calculus/__init__.py
calculus/base_1.py
calculus/test/__init__.py
calculus/test/test_base_1.py
```

To run the tests, you must ensure you are able to load them. Make sure you are in the directory that the `calculus` folder is in, if you run `ls` or `dir` you should see the folder. You can test that you can import the `calculus` package by running `python -c import calculus`. If it reports an error ("No module named calculus"), double check you are in the correct directory.

Run `python -m twisted.trial calculus.test.test_base_1` from the command line when you are in the directory containing the `calculus` directory.

You should see the following output (though your files are probably not in `/tmp`):

```
$ python -m twisted.trial calculus.test.test_base_1
calculus.test.test_base_1
  CalculationTestCase
    test_add ...                                                          [FAIL]
    test_divide ...                                                       [FAIL]
    test_multiply ...                                                     [FAIL]
    test_subtract ...                                                     [FAIL]

===============================================================================
```

```
[FAIL]
Traceback (most recent call last):
  File "/tmp/calculus/test/test_base_1.py", line 8, in test_add
    self.assertEqual(result, 11)
twisted.trial.unittest.FailTest: not equal:
a = None
b = 11


calculus.test.test_base_1.CalculationTestCase.test_add
===============================================================================
[FAIL]
Traceback (most recent call last):
  File "/tmp/calculus/test/test_base_1.py", line 23, in test_divide
    self.assertEqual(result, 2)
twisted.trial.unittest.FailTest: not equal:
a = None
b = 2


calculus.test.test_base_1.CalculationTestCase.test_divide
===============================================================================
[FAIL]
Traceback (most recent call last):
  File "/tmp/calculus/test/test_base_1.py", line 18, in test_multiply
    self.assertEqual(result, 60)
twisted.trial.unittest.FailTest: not equal:
a = None
b = 60


calculus.test.test_base_1.CalculationTestCase.test_multiply
===============================================================================
[FAIL]
Traceback (most recent call last):
  File "/tmp/calculus/test/test_base_1.py", line 13, in test_subtract
    self.assertEqual(result, 4)
twisted.trial.unittest.FailTest: not equal:
a = None
b = 4


calculus.test.test_base_1.CalculationTestCase.test_subtract
-------------------------------------------------------------------------------
Ran 4 tests in 0.042s

FAILED (failures=4)
```

How to interpret this output? You get a list of the individual tests, each followed by its result. By default, failures are printed at the end, but this can be changed with the `-e` (or `--rterrors`) option.

One very useful thing in this output is the fully-qualified name of the failed tests. This appears at the bottom of each =-delimited area of the output. This allows you to copy and paste it to just run a single test you're interested in. In our example, you could run `python -m twisted.trial calculus.test.test_base_1.CalculationTestCase.test_subtract` from the shell.

Note that trial can use different reporters to modify its output. Run `python -m twisted.trial --help-reporters` to see a list of reporters.

The tests can be run by Trial in multiple ways:

- `python -m twisted.trial calculus`: run all the tests for the calculus package.
- `python -m twisted.trial calculus.test`: run using Python's `import` notation.
- `python -m twisted.trial calculus.test.test_base_1`: as above, for a specific test module. You can follow that logic by putting your class name and even a method name to only run those specific tests.
- `python -m twisted.trial --testmodule=calculus/base_1.py`: use the `test-case-name` comment in the first line of `calculus/base_1.py` to find the tests.
- `python -m twisted.trial calculus/test`: run all the tests in the test directory (not recommended).
- `python -m twisted.trial calculus/test/test_base_1.py`: run a specific test file (not recommended).

The first 3 versions using full qualified names are strongly encouraged: they are much more reliable and they allow you to easily be more selective in your test runs.

You'll notice that Trial creates a `_trial_temp` directory in the directory where you run the tests. This has a file called `test.log` which contains the log output of the tests (created using `log.msg` or `log.err` functions). Examine this file if you add logging to your tests.

Making the tests pass

Now that we have a working test framework in place, and our tests are failing (as expected) we can go and try to implement the correct API. We'll do that in a new version of the above base_1 module, `calculus/base_2.py` :

`base_2.py`

```
# -*- test-case-name: calculus.test.test_base_2 -*-

from __future__ import division


class Calculation(object):
    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a - b

    def multiply(self, a, b):
        return a * b

    def divide(self, a, b):
        return a // b
```

We'll also create a new version of test_base_1 which imports and test this new implementation, in `calculus/test_base_2.py`:

`test_base_2.py`

```
from calculus.base_2 import Calculation
from twisted.trial import unittest
```

```
class CalculationTestCase(unittest.TestCase):

    def test_add(self):
        calc = Calculation()
        result = calc.add(3, 8)
        self.assertEqual(result, 11)


    def test_subtract(self):
        calc = Calculation()
        result = calc.subtract(7, 3)
        self.assertEqual(result, 4)


    def test_multiply(self):
        calc = Calculation()
        result = calc.multiply(12, 5)
        self.assertEqual(result, 60)


    def test_divide(self):
        calc = Calculation()
        result = calc.divide(12, 5)
        self.assertEqual(result, 2)
```

is a copy of test_base_1, but with the import changed. Run Trial again as above, and your tests should now pass:

```
$ python -m twisted.trial calculus.test.test_base_2

Running 4 tests.
calculus.test.test_base
  CalculationTestCase
    test_add ...                                                            [OK]
    test_divide ...                                                         [OK]
    test_multiply ...                                                       [OK]
    test_subtract ...                                                       [OK]

-------------------------------------------------------------------------------
Ran 4 tests in 0.067s

PASSED (successes=4)
```

Factoring out common test logic

You'll notice that our test file contains redundant code. Let's get rid of that. Python's unit testing framework allows your test class to define a `setUp` method that is called before *each* test method in the class. This allows you to add attributes to `self` that can be used in test methods. We'll also add a parameterized test method to further simplify the code.

Note that a test class may also provide the counterpart of `setUp` , named `tearDown` , which will be called after *each* test (whether successful or not). `tearDown` is mainly used for post-test cleanup purposes. We will not use `tearDown` until later.

Create `calculus/test/test_base_2b.py` as follows:

`test_base_2b.py`

```
from calculus.base_2 import Calculation
from twisted.trial import unittest



class CalculationTestCase(unittest.TestCase):
    def setUp(self):
        self.calc = Calculation()


    def _test(self, operation, a, b, expected):
        result = operation(a, b)
        self.assertEqual(result, expected)


    def test_add(self):
        self._test(self.calc.add, 3, 8, 11)


    def test_subtract(self):
        self._test(self.calc.subtract, 7, 3, 4)


    def test_multiply(self):
        self._test(self.calc.multiply, 6, 9, 54)


    def test_divide(self):
        self._test(self.calc.divide, 12, 5, 2)
```

Much cleaner, isn't it?

We'll now add some additional error tests. Testing just for successful use of the API is generally not enough, especially if you expect your code to be used by others. Let's make sure the `Calculation` class raises exceptions if someone tries to call its methods with arguments that cannot be converted to integers.

We arrive at `calculus/test/test_base_3.py`:

`test_base_3.py`

```
from calculus.base_3 import Calculation
from twisted.trial import unittest



class CalculationTestCase(unittest.TestCase):
    def setUp(self):
        self.calc = Calculation()


    def _test(self, operation, a, b, expected):
        result = operation(a, b)
        self.assertEqual(result, expected)


    def _test_error(self, operation):
        self.assertRaises(TypeError, operation, "foo", 2)
        self.assertRaises(TypeError, operation, "bar", "egg")
        self.assertRaises(TypeError, operation, [3], [8, 2])
```

```
        self.assertRaises(TypeError, operation, {"e": 3}, {"r": "t"})


    def test_add(self):
        self._test(self.calc.add, 3, 8, 11)


    def test_subtract(self):
        self._test(self.calc.subtract, 7, 3, 4)


    def test_multiply(self):
        self._test(self.calc.multiply, 6, 9, 54)


    def test_divide(self):
        self._test(self.calc.divide, 12, 5, 2)


    def test_errorAdd(self):
        self._test_error(self.calc.add)


    def test_errorSubtract(self):
        self._test_error(self.calc.subtract)


    def test_errorMultiply(self):
        self._test_error(self.calc.multiply)


    def test_errorDivide(self):
        self._test_error(self.calc.divide)
```

We've added four new tests and one general-purpose function, `_test_error` . This function uses the `assertRaises` method, which takes an exception class, a function to run and its arguments, and checks that calling the function on the arguments does indeed raise the given exception.

If you run the above, you'll see that not all tests fail. In Python it's often valid to add and multiply objects of different and even differing types, so the code in the add and multiply tests does not raise an exception and those tests therefore fail. So let's add explicit type conversion to our API class. This brings us to `calculus/base_3.py` :

`base_3.py`

```
# -*- test-case-name: calculus.test.test_base_3 -*-

from __future__ import division


class Calculation(object):
    def _make_ints(self, *args):
        try:
            return [int(arg) for arg in args]
        except ValueError:
            raise TypeError(
                "Couldn't coerce arguments to integers: {}".format(*args))

    def add(self, a, b):
```

```
        a, b = self._make_ints(a, b)
        return a + b

    def subtract(self, a, b):
        a, b = self._make_ints(a, b)
        return a - b

    def multiply(self, a, b):
        a, b = self._make_ints(a, b)
        return a * b

    def divide(self, a, b):
        a, b = self._make_ints(a, b)
        return a // b
```

Here the `_make_ints` helper function tries to convert a list into a list of equivalent integers, and raises a `TypeError` in case the conversion goes wrong.

Note: The `int` conversion can also raise a `TypeError` if passed something of the wrong type, such as a list. We'll just let that exception go by, as `TypeError` is already what we want in case something goes wrong.

Twisted specific testing

Up to this point we've been doing fairly standard Python unit testing. With only a few cosmetic changes (most importantly, directly importing `unittest` instead of using Twisted's unittest version) we could make the above tests run using Python's standard library unit testing framework.

Here we will assume a basic familiarity with Twisted's network I/O, timing, and Deferred APIs. If you haven't already read them, you should read the documentation on *Writing Servers* , *Writing Clients* , and *Deferreds* .

Now we'll get to the real point of this tutorial and take advantage of Trial to test Twisted code.

Testing a protocol

We'll now create a custom protocol to invoke our class from a telnet-like session. We'll remotely call commands with arguments and read back the response. The goal will be to test our network code without creating sockets.

Creating and testing the server

First we'll write the tests, and then explain what they do. The first version of the remote test code is:

`test_remote_1.py`

```
from calculus.remote_1 import RemoteCalculationFactory
from twisted.trial import unittest
from twisted.test import proto_helpers



class RemoteCalculationTestCase(unittest.TestCase):
    def setUp(self):
        factory = RemoteCalculationFactory()
        self.proto = factory.buildProtocol(('127.0.0.1', 0))
```

```
        self.tr = proto_helpers.StringTransport()
        self.proto.makeConnection(self.tr)


    def _test(self, operation, a, b, expected):
        self.proto.dataReceived(u'{} {} {}\r\n'.format(operation, a, b).encode('utf-8
↪'))
        self.assertEqual(int(self.tr.value()), expected)


    def test_add(self):
        return self._test('add', 7, 6, 13)


    def test_subtract(self):
        return self._test('subtract', 82, 78, 4)


    def test_multiply(self):
        return self._test('multiply', 2, 8, 16)


    def test_divide(self):
        return self._test('divide', 14, 3, 4)
```

To fully understand this client, it helps a lot to be comfortable with the Factory/Protocol/Transport pattern used in Twisted.

We first create a protocol factory object. Note that we have yet to see the `RemoteCalculationFactory` class. It is in `calculus/remote_1.py` below. We call `buildProtocol` to ask the factory to build us a protocol object that knows how to talk to our server. We then make a fake network transport, an instance of `twisted.test.proto_helpers.StringTransport` class (note that test packages are generally not part of Twisted's public API;‘‘twisted.test.proto_helpers‘‘ is an exception). This fake transport is the key to the communications. It is used to emulate a network connection without a network. The address and port passed to `buildProtocol` are typically used by the factory to choose to immediately deny remote connections; since we're using a fake transport, we can choose any value that will be acceptable to the factory. In this case the factory just ignores the address, so we don't need to pick anything in particular.

Testing protocols without the use of real network connections is both simple and recommended when testing Twisted code. Even though there are many tests in Twisted that use the network, most good tests don't. The problem with unit tests and networking is that networks aren't reliable. We cannot know that they will exhibit reasonable behavior all the time. This creates intermittent test failures due to network vagaries. Right now we're trying to test our Twisted code, not network reliability. By setting up and using a fake transport, we can write 100% reliable tests. We can also test network failures in a deterministic manner, another important part of your complete test suite.

The final key to understanding this client code is the `_test` method. The call to `dataReceived` simulates data arriving on the network transport. But where does it arrive? It's handed to the `lineReceived` method of the protocol instance (in `calculus/remote_1.py` below). So the client is essentially tricking the server into thinking it has received the operation and the arguments over the network. The server (once again, see below) hands over the work to its `CalculationProxy` object which in turn hands it to its `Calculation` instance. The result is written back via `sendLine` (into the fake string transport object), and is then immediately available to the client, who fetches it with `tr.value()` and checks that it has the expected value. So there's quite a lot going on behind the scenes in the two-line `_test` method above.

Finally , let's see the implementation of this protocol. Put the following into `calculus/remote_1.py` :

`remote_1.py`

```
# -*- test-case-name: calculus.test.test_remote_1 -*-

from twisted.protocols import basic
from twisted.internet import protocol
from calculus.base_3 import Calculation



class CalculationProxy(object):
    def __init__(self):
        self.calc = Calculation()
        for m in ['add', 'subtract', 'multiply', 'divide']:
            setattr(self, 'remote_{}'.format(m), getattr(self.calc, m))



class RemoteCalculationProtocol(basic.LineReceiver):
    def __init__(self):
        self.proxy = CalculationProxy()


    def lineReceived(self, line):
        op, a, b = line.decode('utf-8').split()
        a = int(a)
        b = int(b)
        op = getattr(self.proxy, 'remote_{}'.format(op))
        result = op(a, b)
        self.sendLine(str(result).encode('utf-8'))



class RemoteCalculationFactory(protocol.Factory):
    protocol = RemoteCalculationProtocol



def main():
    from twisted.internet import reactor
    from twisted.python import log
    import sys
    log.startLogging(sys.stdout)
    reactor.listenTCP(0, RemoteCalculationFactory())
    reactor.run()


if __name__ == "__main__":
    main()
```

As mentioned, this server creates a protocol that inherits from basic.LineReceiver , and then a factory that uses it as protocol. The only trick is the `CalculationProxy` object, which calls `Calculation` methods through `remote_*` methods. This pattern is used frequently in Twisted, because it is very explicit about what methods you are making accessible.

If you run this test (`python -m twisted.trial calculus.test.test_remote_1`), everything should be fine. You can also run a server to test it with a telnet client. To do that, call `python calculus/remote_1.py` . You should have the following output:

```
2008-04-25 10:53:27+0200 [-] Log opened.
2008-04-25 10:53:27+0200 [-] __main__.RemoteCalculationFactory starting on 46194
2008-04-25 10:53:27+0200 [-] Starting factory <__main__.RemoteCalculationFactory␣
↪instance at 0x846a0cc>
```

46194 is replaced by a random port. You can then call telnet on it:

```
$ telnet localhost 46194
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
add 4123 9423
13546
```

It works!

Creating and testing the client

Of course, what we build is not particularly useful for now: we'll now build a client for our server, to be able to use it inside a Python program. And it will serve our next purpose.

Create `calculus/test/test_client_1.py`:

`test_client_1.py`

```
from calculus.client_1 import RemoteCalculationClient
from twisted.trial import unittest
from twisted.test import proto_helpers



class ClientCalculationTestCase(unittest.TestCase):
    def setUp(self):
        self.tr = proto_helpers.StringTransport()
        self.proto = RemoteCalculationClient()
        self.proto.makeConnection(self.tr)


    def _test(self, operation, a, b, expected):
        d = getattr(self.proto, operation)(a, b)
        self.assertEqual(
            self.tr.value(),
            u'{} {} {}\r\n'.format(operation, a, b).encode('utf-8')
        )
        self.tr.clear()
        d.addCallback(self.assertEqual, expected)
        self.proto.dataReceived(u"{}\r\n".format(expected,).encode('utf-8'))
        return d


    def test_add(self):
        return self._test('add', 7, 6, 13)


    def test_subtract(self):
        return self._test('subtract', 82, 78, 4)
```

```
    def test_multiply(self):
        return self._test('multiply', 2, 8, 16)


    def test_divide(self):
        return self._test('divide', 14, 3, 4)
```

It's really symmetric to the server test cases. The only tricky part is that we don't use a client factory. We're lazy, and it's not very useful in the client part, so we instantiate the protocol directly.

Incidentally, we have introduced a very important concept here: the tests now return a Deferred object, and the assertion is done in a callback. When a test returns a Deferred, the reactor is run until the Deferred fires and its callbacks run. The important thing to do here is to **not forget to return the Deferred** . If you do, your tests will pass even if nothing is asserted. That's also why it's important to make tests fail first: if your tests pass whereas you know they shouldn't, there is a problem in your tests.

We'll now add the remote client class to produce `calculus/client_1.py` :

`client_1.py`

```
# -*- test-case-name: calculus.test.test_client_1 -*-

from twisted.protocols import basic
from twisted.internet import defer



class RemoteCalculationClient(basic.LineReceiver):
    def __init__(self):
        self.results = []


    def lineReceived(self, line):
        d = self.results.pop(0)
        d.callback(int(line))


    def _sendOperation(self, op, a, b):
        d = defer.Deferred()
        self.results.append(d)
        line = u"{} {} {}".format(op, a, b).encode('utf-8')
        self.sendLine(line)
        return d


    def add(self, a, b):
        return self._sendOperation("add", a, b)


    def subtract(self, a, b):
        return self._sendOperation("subtract", a, b)


    def multiply(self, a, b):
        return self._sendOperation("multiply", a, b)
```

```
    def divide(self, a, b):
        return self._sendOperation("divide", a, b)
```

More good practices

Testing scheduling

When testing code that involves the passage of time, waiting e.g. for a two hour timeout to occur in a test is not very realistic. Twisted provides a solution to this, the Clock class that allows one to simulate the passage of time.

As an example we'll test the code for client request timeout: since our client uses TCP it can hang for a long time (firewall, connectivity problems, etc...). So generally we need to implement timeouts on the client side. Basically it's just that we send a request, don't receive a response and expect a timeout error to be triggered after a certain duration.

test_client_2.py

```
from calculus.client_2 import RemoteCalculationClient, ClientTimeoutError

from twisted.internet import task
from twisted.trial import unittest
from twisted.test import proto_helpers



class ClientCalculationTestCase(unittest.TestCase):
    def setUp(self):
        self.tr = proto_helpers.StringTransportWithDisconnection()
        self.clock = task.Clock()
        self.proto = RemoteCalculationClient()
        self.tr.protocol = self.proto
        self.proto.callLater = self.clock.callLater
        self.proto.makeConnection(self.tr)


    def _test(self, operation, a, b, expected):
        d = getattr(self.proto, operation)(a, b)
        self.assertEqual(
            self.tr.value(),
            u'{} {} {}\r\n'.format(operation, a, b).encode('utf-8')
        )
        self.tr.clear()
        d.addCallback(self.assertEqual, expected)
        self.proto.dataReceived(u"{}\r\n".format(expected).encode('utf-8'))
        return d


    def test_add(self):
        return self._test('add', 7, 6, 13)


    def test_subtract(self):
        return self._test('subtract', 82, 78, 4)


    def test_multiply(self):
        return self._test('multiply', 2, 8, 16)
```

```
    def test_divide(self):
        return self._test('divide', 14, 3, 4)


    def test_timeout(self):
        d = self.proto.add(9, 4)
        self.assertEqual(self.tr.value(), b'add 9 4\r\n')
        self.clock.advance(self.proto.timeOut)
        return self.assertFailure(d, ClientTimeoutError)
```

What happens here? We instantiate our protocol as usual, the only trick is to create the clock, and assign `proto.callLater` to `clock.callLater` . Thus, every `callLater` call in the protocol will finish before `clock.advance()` returns.

In the new test (test_timeout), we call `clock.advance` , that simulates an advance in time (logically it's similar to a `time.sleep` call). And we just have to verify that our Deferred got a timeout error.

Let's implement that in our code.

`client_2.py`

```
# -*- test-case-name: calculus.test.test_client_2 -*-

from twisted.protocols import basic
from twisted.internet import defer, reactor



class ClientTimeoutError(Exception):
    pass



class RemoteCalculationClient(basic.LineReceiver):

    callLater = reactor.callLater
    timeOut = 60

    def __init__(self):
        self.results = []


    def lineReceived(self, line):
        d, callID = self.results.pop(0)
        callID.cancel()
        d.callback(int(line))


    def _cancel(self, d):
        d.errback(ClientTimeoutError())


    def _sendOperation(self, op, a, b):
        d = defer.Deferred()
        callID = self.callLater(self.timeOut, self._cancel, d)
        self.results.append((d, callID))
        line = u"{} {} {}".format(op, a, b).encode('utf-8')
```

```
        self.sendLine(line)
        return d


    def add(self, a, b):
        return self._sendOperation("add", a, b)


    def subtract(self, a, b):
        return self._sendOperation("subtract", a, b)


    def multiply(self, a, b):
        return self._sendOperation("multiply", a, b)


    def divide(self, a, b):
        return self._sendOperation("divide", a, b)
```

If everything completed successfully, it is important to remember to cancel the `DelayedCall` returned by `callLater`.

Cleaning up after tests

This chapter is mainly intended for people who want to have sockets or processes created in their tests. If it's still not obvious, you must try to avoid using them, because it ends up with a lot of problems, one of them being intermittent failures. And intermittent failures are the plague of automated tests.

To actually test that, we'll launch a server with our protocol.

`test_remote_2.py`

```
from calculus.remote_1 import RemoteCalculationFactory
from calculus.client_2 import RemoteCalculationClient

from twisted.trial import unittest
from twisted.internet import reactor, protocol



class RemoteRunCalculationTestCase(unittest.TestCase):

    def setUp(self):
        factory = RemoteCalculationFactory()
        self.port = reactor.listenTCP(0, factory, interface="127.0.0.1")
        self.client = None


    def tearDown(self):
        if self.client is not None:
            self.client.transport.loseConnection()
        return self.port.stopListening()


    def _test(self, op, a, b, expected):
        creator = protocol.ClientCreator(reactor, RemoteCalculationClient)
        def cb(client):
```

```
            self.client = client
            return getattr(self.client, op)(a, b
                ).addCallback(self.assertEqual, expected)
        return creator.connectTCP('127.0.0.1', self.port.getHost().port
            ).addCallback(cb)


    def test_add(self):
        return self._test("add", 5, 9, 14)


    def test_subtract(self):
        return self._test("subtract", 47, 13, 34)


    def test_multiply(self):
        return self._test("multiply", 7, 3, 21)


    def test_divide(self):
        return self._test("divide", 84, 10, 8)
```

Recent versions of Trial will fail loudly if you remove the `stopListening` call, which is good.

Also, you should be aware that `tearDown` will be called in any case, after success or failure. So don't expect every object you created in the test method to be present, because your tests may have failed in the middle.

Trial also has a `addCleanup` method, which makes these kind of cleanups easy and removes the need for `tearDown`. For example, you could remove the code in `_test` this way:

```
def setUp(self):
    factory = RemoteCalculationFactory()
    self.port = reactor.listenTCP(0, factory, interface="127.0.0.1")
    self.addCleanup(self.port.stopListening)

def _test(self, op, a, b, expected):
    creator = protocol.ClientCreator(reactor, RemoteCalculationClient)
    def cb(client):
        self.addCleanup(self.client.transport.loseConnection)
        return getattr(client, op)(a, b).addCallback(self.assertEqual, expected)
    return creator.connectTCP('127.0.0.1', self.port.getHost().port).addCallback(cb)
```

This removes the need of a `tearDown` method, and you don't have to check for the value of self.client: you only call addCleanup when the client is created.

Handling logged errors

Currently, if you send an invalid command or invalid arguments to our server, it logs an exception and closes the connection. This is a perfectly valid behavior, but for the sake of this tutorial, we want to return an error to the user if they send invalid operators, and log any errors on server side. So we'll want a test like this:

```
def test_invalidParameters(self):
    self.proto.dataReceived('add foo bar\r\n')
    self.assertEqual(self.tr.value(), "error\r\n")
```

`remote_2.py`

```
# -*- test-case-name: calculus.test.test_remote_1 -*-

from twisted.protocols import basic
from twisted.internet import protocol
from twisted.python import log
from calculus.base_3 import Calculation



class CalculationProxy(object):
    def __init__(self):
        self.calc = Calculation()
        for m in ['add', 'subtract', 'multiply', 'divide']:
            setattr(self, 'remote_{}'.format(m), getattr(self.calc, m))



class RemoteCalculationProtocol(basic.LineReceiver):
    def __init__(self):
        self.proxy = CalculationProxy()


    def lineReceived(self, line):
        op, a, b = line.decode('utf-8').split()
        op = getattr(self.proxy, 'remote_{}'.format(op,))
        try:
            result = op(a, b)
        except TypeError:
            log.err()
            self.sendLine(b"error")
        else:
            self.sendLine(str(result).encode('utf-8'))



class RemoteCalculationFactory(protocol.Factory):
    protocol = RemoteCalculationProtocol



def main():
    from twisted.internet import reactor
    from twisted.python import log
    import sys
    log.startLogging(sys.stdout)
    reactor.listenTCP(0, RemoteCalculationFactory())
    reactor.run()


if __name__ == "__main__":
    main()
```

If you try something like that, it will not work. Here is the output you should have:

```
$ python -m twisted.trial calculus.test.test_remote_3.RemoteCalculationTestCase.test_
↪invalidParameters
calculus.test.test_remote_3
  RemoteCalculationTestCase
```

```
    test_invalidParameters ...                                              [ERROR]

===============================================================================
[ERROR]: calculus.test.test_remote_3.RemoteCalculationTestCase.test_invalidParameters

Traceback (most recent call last):
  File "/tmp/calculus/remote_2.py", line 27, in lineReceived
    result = op(a, b)
  File "/tmp/calculus/base_3.py", line 11, in add
    a, b = self._make_ints(a, b)
  File "/tmp/calculus/base_3.py", line 8, in _make_ints
    raise TypeError
exceptions.TypeError:
-------------------------------------------------------------------------------
Ran 1 tests in 0.004s

FAILED (errors=1)
```

At first, you could think there is a problem, because you catch this exception. But in fact Trial doesn't let you do that without controlling it: you must expect logged errors and clean them. To do that, you have to use the `flushLoggedErrors` method. You call it with the exception you expect, and it returns the list of exceptions logged since the start of the test. Generally, you'll want to check that this list has the expected length, or possibly that each exception has an expected message. We do the former in our test:

`test_remote_3.py`

```
from calculus.remote_2 import RemoteCalculationFactory
from twisted.trial import unittest
from twisted.test import proto_helpers



class RemoteCalculationTestCase(unittest.TestCase):
    def setUp(self):
        factory = RemoteCalculationFactory()
        self.proto = factory.buildProtocol(('127.0.0.1', 0))
        self.tr = proto_helpers.StringTransport()
        self.proto.makeConnection(self.tr)


    def _test(self, operation, a, b, expected):
        self.proto.dataReceived(
            u'{} {} {}\r\n'.format(operation, a, b).encode('utf-8')
        )
        self.assertEqual(int(self.tr.value()), expected)


    def test_add(self):
        return self._test('add', 7, 6, 13)


    def test_subtract(self):
        return self._test('subtract', 82, 78, 4)


    def test_multiply(self):
        return self._test('multiply', 2, 8, 16)
```

```
    def test_divide(self):
        return self._test('divide', 14, 3, 4)


    def test_invalidParameters(self):
        self.proto.dataReceived(b'add foo bar\r\n')
        self.assertEqual(self.tr.value(), b"error\r\n")
        errors = self.flushLoggedErrors(TypeError)
        self.assertEqual(len(errors), 1)
```

Resolve a bug

A bug was left over during the development of the timeout (probably several bugs, but that's not the point), concerning the reuse of the protocol when you got a timeout: the connection is not dropped, so you can get timeout forever. Generally a user will come to you saying "I have this strange problem on my crappy network. It seems you could solve it with doing XXX at YYY."

Actually, this bug can be corrected several ways. But if you correct it without adding tests, one day you'll face a big problem: regression. So the first step is adding a failing test.

test_client_3.py

```
from calculus.client_3 import RemoteCalculationClient, ClientTimeoutError

from twisted.internet import task
from twisted.trial import unittest
from twisted.test import proto_helpers



class ClientCalculationTestCase(unittest.TestCase):
    def setUp(self):
        self.tr = proto_helpers.StringTransportWithDisconnection()
        self.clock = task.Clock()
        self.proto = RemoteCalculationClient()
        self.tr.protocol = self.proto
        self.proto.callLater = self.clock.callLater
        self.proto.makeConnection(self.tr)


    def _test(self, operation, a, b, expected):
        d = getattr(self.proto, operation)(a, b)
        self.assertEqual(
            self.tr.value(),
            u'{} {} {}\r\n'.format(operation, a, b).encode('utf-8')
        )
        self.tr.clear()
        d.addCallback(self.assertEqual, expected)
        self.proto.dataReceived(u"{}\r\n".format(expected).encode('utf-8'))
        return d


    def test_add(self):
        return self._test('add', 7, 6, 13)
```

```
    def test_subtract(self):
        return self._test('subtract', 82, 78, 4)


    def test_multiply(self):
        return self._test('multiply', 2, 8, 16)


    def test_divide(self):
        return self._test('divide', 14, 3, 4)


    def test_timeout(self):
        d = self.proto.add(9, 4)
        self.assertEqual(self.tr.value(), b'add 9 4\r\n')
        self.clock.advance(self.proto.timeOut)
        return self.assertFailure(d, ClientTimeoutError)


    def test_timeoutConnectionLost(self):
        called = []
        def lost(arg):
            called.append(True)
        self.proto.connectionLost = lost

        d = self.proto.add(9, 4)
        self.assertEqual(self.tr.value(), b'add 9 4\r\n')
        self.clock.advance(self.proto.timeOut)

        def check(ignore):
            self.assertEqual(called, [True])
        return self.assertFailure(d, ClientTimeoutError).addCallback(check)
```

What have we done here?

- We switched to StringTransportWithDisconnection. This transport manages `loseConnection` and forwards it to its protocol.
- We assign the protocol to the transport via the `protocol` attribute.
- We check that after a timeout our connection has closed.

For doing that, we then use the `TimeoutMixin` class, that does almost everything we want. The great thing is that it almost changes nothing to our class.

`client_3.py`

```
# -*- test-case-name: calculus.test.test_client -*-

from twisted.protocols import basic, policies
from twisted.internet import defer



class ClientTimeoutError(Exception):
    pass



class RemoteCalculationClient(basic.LineReceiver, policies.TimeoutMixin):
```

```
    def __init__(self):
        self.results = []
        self._timeOut = 60

    def lineReceived(self, line):
        self.setTimeout(None)
        d = self.results.pop(0)
        d.callback(int(line))


    def timeoutConnection(self):
        for d in self.results:
            d.errback(ClientTimeoutError())
        self.transport.loseConnection()


    def _sendOperation(self, op, a, b):
        d = defer.Deferred()
        self.results.append(d)
        line = u"{} {} {}".format(op, a, b).encode('utf-8')
        self.sendLine(line)
        self.setTimeout(self._timeOut)
        return d


    def add(self, a, b):
        return self._sendOperation("add", a, b)


    def subtract(self, a, b):
        return self._sendOperation("subtract", a, b)


    def multiply(self, a, b):
        return self._sendOperation("multiply", a, b)


    def divide(self, a, b):
        return self._sendOperation("divide", a, b)
```

Testing Deferreds without the reactor

Above we learned about returning Deferreds from test methods in order to make assertions about their results, or side-effects that only happen after they fire. This can be useful, but we don't actually need the feature in this example. Because we were careful to use `Clock` , we don't need the global reactor to run in our tests. Instead of returning the Deferred with a callback attached to it which performs the necessary assertions, we can use a testing helper, successResultOf (and the corresponding error-case helper failureResultOf), to extract its result and make assertions against it directly. Compared to returning a Deferred, this avoids the problem of forgetting to return the Deferred, improves the stack trace reported when the assertion fails, and avoids the complexity of using global reactor (which, for example, may then require cleanup).

`test_client_4.py`

```
from calculus.client_3 import RemoteCalculationClient, ClientTimeoutError
```

```
from twisted.internet import task
from twisted.trial import unittest
from twisted.test import proto_helpers



class ClientCalculationTestCase(unittest.TestCase):
    def setUp(self):
        self.tr = proto_helpers.StringTransportWithDisconnection()
        self.clock = task.Clock()
        self.proto = RemoteCalculationClient()
        self.tr.protocol = self.proto
        self.proto.callLater = self.clock.callLater
        self.proto.makeConnection(self.tr)


    def _test(self, operation, a, b, expected):
        d = getattr(self.proto, operation)(a, b)
        self.assertEqual(self.tr.value(), u'{} {} {}\r\n'.format(operation, a, b).
↪encode('utf-8'))
        self.tr.clear()
        self.proto.dataReceived(u"{}\r\n".format(expected).encode('utf-8'))
        self.assertEqual(expected, self.successResultOf(d))


    def test_add(self):
        self._test('add', 7, 6, 13)


    def test_subtract(self):
        self._test('subtract', 82, 78, 4)


    def test_multiply(self):
        self._test('multiply', 2, 8, 16)


    def test_divide(self):
        self._test('divide', 14, 3, 4)


    def test_timeout(self):
        d = self.proto.add(9, 4)
        self.assertEqual(self.tr.value(), b'add 9 4\r\n')
        self.clock.advance(self.proto.timeOut)
        self.failureResultOf(d).trap(ClientTimeoutError)


    def test_timeoutConnectionLost(self):
        called = []
        def lost(arg):
            called.append(True)
        self.proto.connectionLost = lost

        d = self.proto.add(9, 4)
        self.assertEqual(self.tr.value(), b'add 9 4\r\n')
        self.clock.advance(self.proto.timeOut)
```

```
        def check(ignore):
            self.assertEqual(called, [True])
        self.failureResultOf(d).trap(ClientTimeoutError)
        self.assertEqual(called, [True])
```

This version of the code makes the same assertions, but no longer returns any Deferreds from any test methods. Instead of making assertions about the result of the Deferred in a callback, it makes the assertions as soon as it *knows* the Deferred is supposed to have a result (in the `_test` method and in `test_timeout` and `test_timeoutConnectionLost`). The possibility of *knowing* exactly when a Deferred is supposed to have a test is what makes `successResultOf` useful in unit testing, but prevents it from being applicable to non-testing purposes.

`successResultOf` will raise an exception (failing the test) if the `Deferred` passed to it does not have a result, or has a failure result. Similarly, `failureResultOf` will raise an exception (also failing the test) if the `Deferred` passed to it does not have a result, or has a success result. There is a third helper method for testing the final case, assertNoResult , which only raises an exception (failing the test) if the `Deferred` passed to it *has* a result (either success or failure).

Dropping into a debugger

In the course of writing and running your tests, it is often helpful to employ the use of a debugger. This can be particularly helpful in tracking down where the source of a troublesome bug is in your code. Python's standard library includes a debugger in the form of the pdb module. Running your tests with `pdb` is as simple as invoking twisted with the `--debug` option, which will start `pdb` at the beginning of the execution of your test suite.

Trial also provides a `--debugger` option which can run your test suite using another debugger instead. To specify a debugger other than `pdb` , pass in the fully-qualified name of an object that provides the same interface as `pdb` . Most third-party debuggers tend to implement an interface similar to `pdb` , or at least provide a wrapper object that does. For example, invoking Trial with the extra arguments `-debug --debugger pudb` will open the PuDB debugger instead, provided it is properly installed.

Code coverage

Code coverage is one of the aspects of software testing that shows how much your tests cross (cover) the code of your program. There are different kinds of measures: path coverage, condition coverage, statement coverage... We'll only consider statement coverage here, whether a line has been executed or not.

Trial has an option to generate the statement coverage of your tests. This option is –coverage. It creates a coverage directory in _trial_temp, with a file .cover for every module used during the tests. The ones interesting for us are calculus.base.cover and calculus.remote.cover. Each line starts with a counter showing how many times the line was executed during the tests, or the marker '>>>>>>' if the line was not covered. If you went through the whole tutorial to this point, you should have complete coverage :).

Again, this is only another useful pointer, but it doesn't mean your code is perfect: your tests should consider every possible input and output, to get **full** coverage (condition, path, etc.) as well .

Conclusion

So what did you learn in this document?

- How to use the Trial command-line tool to run your tests
- How to use string transports to test individual clients and servers without creating sockets
- If you really want to create sockets, how to cleanly do it so that it doesn't have bad side effects

- And some small tips you can't live without.

If one of the topics still looks cloudy to you, please give us your feedback! You can file tickets to improve this document - learn how to contribute on the Twisted web site.

Twisted from Scratch, or The Evolution of Finger

The Evolution of Finger: building a simple finger service

Introduction

This is the first part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* .

If you're not familiar with 'finger' it's probably because it's not used as much nowadays as it used to be. Basically, if you run `finger nail` or `finger nail@example.com` the target computer spits out some information about the user named `nail` . For instance:

```
Login: nail                             Name: Nail Sharp
Directory: /home/nail                   Shell: /usr/bin/sh
Last login Wed Mar 31 18:32 2004 (PST)
New mail received Thu Apr  1 10:50 2004 (PST)
     Unread since Thu Apr  1 10:50 2004 (PST)
No Plan.
```

If the target computer does not have the `fingerd` *daemon* running you'll get a "Connection Refused" error. Paranoid sysadmins keep `fingerd` off or limit the output to hinder crackers and harassers. The above format is the standard `fingerd` default, but an alternate implementation can output anything it wants, such as automated responsibility status for everyone in an organization. You can also define pseudo "users", which are essentially keywords.

This portion of the tutorial makes use of factories and protocols as introduced in the *Writing a TCP Server howto* and deferreds as introduced in *Using Deferreds* and *Generating Deferreds* . Services and applications are discussed in *Using the Twisted Application Framework* .

By the end of this section of the tutorial, our finger server will answer TCP finger requests on port 1079, and will read data from the web.

Refuse Connections

`finger01.py`

```
from twisted.internet import reactor
reactor.run()
```

This example only runs the reactor. It will consume almost no CPU resources. As it is not listening on any port, it can't respond to network requests — nothing at all will happen until we interrupt the program. At this point if you run `finger nail` or `telnet localhost 1079` , you'll get a "Connection refused" error since there's no daemon running to respond. Not very useful, perhaps — but this is the skeleton inside which the Twisted program will grow.

As implied above, at various points in this tutorial you'll want to observe the behavior of the server being developed. Unless you have a finger program which can use an alternate port, the easiest way to do this is with a telnet client. `telnet localhost 1079` will connect to the local host on port 1079, where a finger server will eventually be listening.

The Reactor

You don't call Twisted, Twisted calls you. The reactor is Twisted's main event loop, similar to the main loop in other toolkits available in Python (Qt, wx, and Gtk). There is exactly one reactor in any running Twisted application. Once started it loops over and over again, responding to network events and making scheduled calls to code.

Note that there are actually several different reactors to choose from; `from twisted.internet import reactor` returns the current reactor. If you haven't chosen a reactor class yet, it automatically chooses the default. See the *Reactor Basics HOWTO* for more information.

Do Nothing

`finger02.py`

```
from twisted.internet import protocol, reactor, endpoints

class FingerProtocol(protocol.Protocol):
    pass

class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol

fingerEndpoint = endpoints.serverFromString(reactor, "tcp:1079")
fingerEndpoint.listen(FingerFactory())
reactor.run()
```

Here we use `endpoints.serverFromString` to create a Twisted endpoint. An endpoint is a Twisted concept that encapsulates one end of a connection. There are different endpoints for clients and servers. One of the great advantages of endpoints is that they can be described textually using a kind of domain-specific language. For example, here, we ask Twisted to create a TCP endpoint for a server using the string `"tcp:1079"`. That, along with the call to `serverFromString`, tells Twisted to look for a TCP endpoint, and pass it the port 1079. The endpoint returned from that function can then have the `listen()` method invoked on it, which causes Twisted to start listening on port 1079. (The number 1079 is a reminder that eventually we want to run on port 79, the standard port for finger servers.) For more detail on endpoints, check out the *Getting Connected With Endpoints*.

The factory given to the `listen()` method, `FingerFactory` , is used to handle incoming requests on that port. Specifically, for each request, the reactor calls the factory's `buildProtocol` method, which in this case causes `FingerProtocol` to be instantiated. Since the protocol defined here does not actually respond to any events, connections to 1079 will be accepted, but the input ignored.

A Factory is the proper place for data that you want to make available to the protocol instances, since the protocol instances are garbage collected when the connection is closed.

Drop Connections

`finger03.py`

```
from twisted.internet import protocol, reactor, endpoints

class FingerProtocol(protocol.Protocol):
    def connectionMade(self):
        self.transport.loseConnection()

class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol
```

```
fingerEndpoint = endpoints.serverFromString(reactor, "tcp:1079")
fingerEndpoint.listen(FingerFactory())
reactor.run()
```

Here we add to the protocol the ability to respond to the event of beginning a connection — by terminating it. Perhaps not an interesting behavior, but it is already close to behaving according to the letter of the standard finger protocol. After all, there is no requirement to send any data to the remote connection in the standard. The only problem, as far as the standard is concerned, is that we terminate the connection too soon. A client which is slow enough will see his `send()` of the username result in an error.

Read Username, Drop Connections

`finger04.py`

```
from twisted.internet import protocol, reactor, endpoints
from twisted.protocols import basic

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.transport.loseConnection()

class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol

fingerEndpoint = endpoints.serverFromString(reactor, "tcp:1079")
fingerEndpoint.listen(FingerFactory())
reactor.run()
```

Here we make `FingerProtocol` inherit from LineReceiver , so that we get data-based events on a line-by-line basis. We respond to the event of receiving the line with shutting down the connection.

If you use a telnet client to interact with this server, the result will look something like this:

```
$ telnet localhost 1079
Trying 127.0.0.1...
Connected to localhost.localdomain.
alice
Connection closed by foreign host.
```

Congratulations, this is the first standard-compliant version of the code. However, usually people actually expect some data about users to be transmitted.

Read Username, Output Error, Drop Connections

`finger05.py`

```
from twisted.internet import protocol, reactor, endpoints
from twisted.protocols import basic

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.transport.write(b"No such user\r\n")
        self.transport.loseConnection()
```

```
class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol

fingerEndpoint = endpoints.serverFromString(reactor, "tcp:1079")
fingerEndpoint.listen(FingerFactory())
reactor.run()
```

Finally, a useful version. Granted, the usefulness is somewhat limited by the fact that this version only prints out a "No such user" message. It could be used for devastating effect in honey-pots (decoy servers), of course.

Output From Empty Factory

finger06.py

```
# Read username, output from empty factory, drop connections

from twisted.internet import protocol, reactor, endpoints
from twisted.protocols import basic

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.transport.write(self.factory.getUser(user)+ b"\r\n")
        self.transport.loseConnection()

class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol

    def getUser(self, user):
        return b"No such user"

fingerEndpoint = endpoints.serverFromString(reactor, "tcp:1079")
fingerEndpoint.listen(FingerFactory())
reactor.run()
```

The same behavior, but finally we see what usefulness the factory has: as something that does not get constructed for every connection, it can be in charge of the user database. In particular, we won't have to change the protocol if the user database back-end changes.

Output from Non-empty Factory

finger07.py

```
# Read username, output from non-empty factory, drop connections

from twisted.internet import protocol, reactor, endpoints
from twisted.protocols import basic

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.transport.write(self.factory.getUser(user) + b"\r\n")
        self.transport.loseConnection()

class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol
```

```
    def __init__(self, users):
        self.users = users

    def getUser(self, user):
        return self.users.get(user, b"No such user")

fingerEndpoint = endpoints.serverFromString(reactor, "tcp:1079")
fingerEndpoint.listen(FingerFactory({ b'moshez' : b'Happy and well'}))
reactor.run()
```

Finally, a really useful finger database. While it does not supply information about logged in users, it could be used to distribute things like office locations and internal office numbers. As hinted above, the factory is in charge of keeping the user database: note that the protocol instance has not changed. This is starting to look good: we really won't have to keep tweaking our protocol.

Use Deferreds

finger08.py

```
# Read username, output from non-empty factory, drop connections
# Use deferreds, to minimize synchronicity assumptions

from twisted.internet import protocol, reactor, defer, endpoints
from twisted.protocols import basic

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        d = self.factory.getUser(user)

        def onError(err):
            return 'Internal error in server'
        d.addErrback(onError)

        def writeResponse(message):
            self.transport.write(message + b'\r\n')
            self.transport.loseConnection()
        d.addCallback(writeResponse)

class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol

    def __init__(self, users):
        self.users = usrs

    def getUser(self, user):
        return defer.succeed(self.users.get(user, b"No such user"))

fingerEndpoint = endpoints.serverFromString(reactor, "tcp:1079")
fingerEndpoint.listen(FingerFactory({b'moshez': b'Happy and well'}))
reactor.run()
```

But, here we tweak it just for the hell of it. Yes, while the previous version worked, it did assume the result of getUser is always immediately available. But what if instead of an in-memory database, we would have to fetch the result from a remote Oracle server? By allowing getUser to return a Deferred, we make it easier for the data to be retrieved asynchronously so that the CPU can be used for other tasks in the meanwhile.

As described in the *Deferred HOWTO* , Deferreds allow a program to be driven by events. For instance, if one task in a program is waiting on data, rather than have the CPU (and the program!) idly waiting for that data (a process normally called 'blocking'), the program can perform other operations in the meantime, and waits for some signal that data is ready to be processed before returning to that process.

In brief, the code in `FingerFactory` above creates a Deferred, to which we start to attach *callbacks* . The deferred action in `FingerFactory` is actually a fast-running expression consisting of one dictionary method, `get` . Since this action can execute without delay, `FingerFactory.getUser` uses `defer.succeed` to create a Deferred which already has a result, meaning its return value will be passed immediately to the first callback function, which turns out to be `FingerProtocol.writeResponse` . We've also defined an *errback* (appropriately named `FingerProtocol.onError`) that will be called instead of `writeResponse` if something goes wrong.

Run 'finger' Locally

`finger09.py`

```
# Read username, output from factory interfacing to OS, drop connections

from twisted.internet import protocol, reactor, defer, utils, endpoints
from twisted.protocols import basic

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        d = self.factory.getUser(user)

        def onError(err):
            return b'Internal error in server'
        d.addErrback(onError)

        def writeResponse(message):
            self.transport.write(message + b'\r\n')
            self.transport.loseConnection()
        d.addCallback(writeResponse)

class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol

    def getUser(self, user):
        return utils.getProcessOutput(b"finger", [user])

fingerEndpoint = endpoints.serverFromString(reactor, "tcp:1079")
fingerEndpoint.listen(FingerFactory())
reactor.run()
```

This example also makes use of a Deferred. `twisted.internet.utils.getProcessOutput` is a non-blocking version of Python's `commands.getoutput` : it runs a shell command (`finger` , in this case) and captures its standard output. However, `getProcessOutput` returns a Deferred instead of the output itself. Since `FingerProtocol.lineReceived` is already expecting a Deferred to be returned by `getUser` , it doesn't need to be changed, and it returns the standard output as the finger result.

Note that in this case the shell's built-in `finger` command is simply run with whatever arguments it is given. This is probably insecure, so you probably don't want a real server to do this without a lot more validation of the user input. This will do exactly what the standard version of the finger server does.

Read Status from the Web

The web. That invention which has infiltrated homes around the world finally gets through to our invention. In this case we use the built-in Twisted web client via `twisted.web.client.getPage`, a non-blocking version of Python's `urllib2.urlopen(URL).read`. Like `getProcessOutput` it returns a Deferred which will be called back with a string, and can thus be used as a drop-in replacement.

Thus, we have examples of three different database back-ends, none of which change the protocol class. In fact, we will not have to change the protocol again until the end of this tutorial: we have achieved, here, one truly usable class.

`finger10.py`

```
# Read username, output from factory interfacing to web, drop connections

from twisted.internet import protocol, reactor, defer, utils, endpoints
from twisted.protocols import basic
from twisted.web import client

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        d = self.factory.getUser(user)

        def onError(err):
            return b'Internal error in server'
        d.addErrback(onError)

        def writeResponse(message):
            self.transport.write(message + b'\r\n')
            self.transport.loseConnection()
        d.addCallback(writeResponse)

class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol

    def __init__(self, prefix):
        self.prefix = prefix

    def getUser(self, user):
        return client.getPage(self.prefix + user)

fingerEndpoint = endpoints.serverFromString(reactor, "tcp:1079")
fingerEndpoint.listen(FingerFactory(prefix=b'http://livejournal.com/~'))
reactor.run()
```

Use Application

Up until now, we faked. We kept using port 1079, because really, who wants to run a finger server with root privileges? Well, the common solution is "privilege shedding" : after binding to the network, become a different, less privileged user. We could have done it ourselves, but Twisted has a built-in way to do it. We will create a snippet as above, but now we will define an application object. That object will have `uid` and `gid` attributes. When running it (later we will see how) it will bind to ports, shed privileges and then run.

Read on to find out how to run this code using the twistd utility.

twistd

This is how to run "Twisted Applications" — files which define an 'application'. A daemon is expected to adhere to certain behavioral standards so that standard tools can stop/start/query them. If a Twisted application is run via twistd, the TWISTed Daemonizer, all this behavioral stuff will be handled for you. twistd does everything a daemon can be expected to — shuts down stdin/stdout/stderr, disconnects from the terminal and can even change runtime directory, or even the root filesystems. In short, it does everything so the Twisted application developer can concentrate on writing his networking code.

```
root% twistd -ny finger11.tac # just like before
root% twistd -y finger11.tac # daemonize, keep pid in twistd.pid
root% twistd -y finger11.tac --pidfile=finger.pid
root% twistd -y finger11.tac --rundir=/
root% twistd -y finger11.tac --chroot=/var
root% twistd -y finger11.tac -l /var/log/finger.log
root% twistd -y finger11.tac --syslog # just log to syslog
root% twistd -y finger11.tac --syslog --prefix=twistedfinger # use given prefix
```

There are several ways to tell twistd where your application is; here we show how it is done using the `application` global variable in a Python source file (a *Twisted Application Configuration* file).

`finger11.tac`

```
# Read username, output from non-empty factory, drop connections
# Use deferreds, to minimize synchronicity assumptions
# Write application. Save in 'finger.tpy'

from twisted.application import service, strports
from twisted.internet import protocol, reactor, defer
from twisted.protocols import basic

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        d = self.factory.getUser(user)

        def onError(err):
            return 'Internal error in server'
        d.addErrback(onError)

        def writeResponse(message):
            self.transport.write(message + b'\r\n')
            self.transport.loseConnection()
        d.addCallback(writeResponse)

class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol

    def __init__(self, users):
        self.users = users

    def getUser(self, user):
        return defer.succeed(self.users.get(user, b"No such user"))

application = service.Application('finger', uid=1, gid=1)
factory = FingerFactory({b'moshez': b'Happy and well'})
strports.service("tcp:79", factory, reactor=reactor).setServiceParent(
    service.IServiceCollection(application))
```

Instead of using `endpoints.serverFromString` as in the above examples, here we are using its application-aware counterpart, `strports.service` . Notice that when it is instantiated, the application object itself does not reference either the protocol or the factory. Any services (such as the one we created with `strports.service`) which have the application as their parent will be started when the application is started by twistd. The application object is more useful for returning an object that supports the IService , IServiceCollection , IProcess , and sob.IPersistable interfaces with the given parameters; we'll be seeing these in the next part of the tutorial. As the parent of the endpoint we opened, the application lets us manage the endpoint.

With the daemon running on the standard finger port, you can test it with the standard finger command: `finger moshez` .

The Evolution of Finger: adding features to the finger service

Introduction

This is the second part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* .

In this section of the tutorial, our finger server will continue to sprout features: the ability for users to set finger announces, and using our finger service to send those announcements on the web, on IRC and over XML-RPC. Resources and XML-RPC are introduced in the Web Applications portion of the *Twisted Web howto* . More examples using twisted.words.protocols.irc can be found in *Writing a TCP Client* and the *Twisted Words examples* .

Setting Message By Local Users

Now that port 1079 is free, maybe we can use it with a different server, one which will let people set their messages. It does no access control, so anyone who can login to the machine can set any message. We assume this is the desired behavior in our case. Testing it can be done by simply:

```
% nc localhost 1079   # or telnet localhost 1079
moshez
Giving a tutorial now, sorry!
^D
```

`finger12.tac`

```
# But let's try and fix setting away messages, shall we?
from twisted.application import service, strports
from twisted.internet import protocol, reactor, defer
from twisted.protocols import basic

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        d = self.factory.getUser(user)

        def onError(err):
            return b'Internal error in server'
        d.addErrback(onError)

        def writeResponse(message):
            self.transport.write(message + b'\r\n')
            self.transport.loseConnection()
        d.addCallback(writeResponse)

class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol
```

```
    def __init__(self, users):
        self.users = users

    def getUser(self, user):
        return defer.succeed(self.users.get(user, b"No such user"))

class FingerSetterProtocol(basic.LineReceiver):
    def connectionMade(self):
        self.lines = []

    def lineReceived(self, line):
        self.lines.append(line)

    def connectionLost(self, reason):
        user = self.lines[0]
        status = self.lines[1]
        self.factory.setUser(user, status)

class FingerSetterFactory(protocol.ServerFactory):
    protocol = FingerSetterProtocol

    def __init__(self, fingerFactory):
        self.fingerFactory = fingerFactory

    def setUser(self, user, status):
        self.fingerFactory.users[user] = status

ff = FingerFactory({b'moshez': b'Happy and well'})
fsf = FingerSetterFactory(ff)

application = service.Application('finger', uid=1, gid=1)
serviceCollection = service.IServiceCollection(application)
strports.service("tcp:79", ff).setServiceParent(serviceCollection)
strports.service("tcp:1079", fsf).setServiceParent(serviceCollection)
```

This program has two protocol-factory-TCPServer pairs, which are both child services of the application. Specifically, the setServiceParent method is used to define the two TCPServer services as children of `application` , which implements IServiceCollection . Both services are thus started with the application.

Use Services to Make Dependencies Sane

The previous version had the setter poke at the innards of the finger factory. This strategy is usually not a good idea: this version makes both factories symmetric by making them both look at a single object. Services are useful for when an object is needed which is not related to a specific network server. Here, we define a common service class with methods that will create factories on the fly. The service also contains methods the factories will depend on.

The factory-creation methods, `getFingerFactory` and `getFingerSetterFactory` , follow this pattern:

1. Instantiate a generic server factory, `twisted.internet.protocol.ServerFactory` .
2. Set the protocol class, just like our factory class would have.
3. Copy a service method to the factory as a function attribute. The function won't have access to the factory's `self` , but that's OK because as a bound method it has access to the service's `self` , which is what it needs. For `getUser` , a custom method defined in the service gets copied. For `setUser` , a standard method of the `users` dictionary is copied.

Thus, we stopped subclassing: the service simply puts useful methods and attributes inside the factories. We are getting better at protocol design: none of our protocol classes had to be changed, and neither will have to change until the end of the tutorial.

As an application service, this new finger service implements the IService interface and can be started and stopped in a standardized manner. We'll make use of this in the next example.

finger13.tac

```
# Fix asymmetry
from twisted.application import service, strports
from twisted.internet import protocol, reactor, defer
from twisted.protocols import basic

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        d = self.factory.getUser(user)

        def onError(err):
            return b'Internal error in server'
        d.addErrback(onError)

        def writeResponse(message):
            self.transport.write(message + b'\r\n')
            self.transport.loseConnection()
        d.addCallback(writeResponse)

class FingerSetterProtocol(basic.LineReceiver):
    def connectionMade(self):
        self.lines = []

    def lineReceived(self, line):
        self.lines.append(line)

    def connectionLost(self,reason):
        user = self.lines[0]
        status = self.lines[1]
        self.factory.setUser(user, status)

class FingerService(service.Service):
    def __init__(self, users):
        self.users = users

    def getUser(self, user):
        return defer.succeed(self.users.get(user, b"No such user"))

    def setUser(self, user, status):
        self.users[user] = status

    def getFingerFactory(self):
        f = protocol.ServerFactory()
        f.protocol = FingerProtocol
        f.getUser = self.getUser
        return f

    def getFingerSetterFactory(self):
        f = protocol.ServerFactory()
        f.protocol = FingerSetterProtocol
        f.setUser = self.setUser
```

```
        return f

application = service.Application('finger', uid=1, gid=1)
f = FingerService({b'moshez': b'Happy and well'})
serviceCollection = service.IServiceCollection(application)
strports.service("tcp:79", f.getFingerFactory()
                 ).setServiceParent(serviceCollection)
strports.service("tcp:1079", f.getFingerSetterFactory()
                 ).setServiceParent(serviceCollection)
```

Most application services will want to use the Service base class, which implements all the generic `IService` behavior.

Read Status File

This version shows how, instead of just letting users set their messages, we can read those from a centrally managed file. We cache results, and every 30 seconds we refresh it. Services are useful for such scheduled tasks.

listings/finger/etc.users

`finger14.tac`

```
# Read from file
from twisted.application import service, strports
from twisted.internet import protocol, reactor, defer
from twisted.protocols import basic

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        d = self.factory.getUser(user)

        def onError(err):
            return b'Internal error in server'
        d.addErrback(onError)

        def writeResponse(message):
            self.transport.write(message + b'\r\n')
            self.transport.loseConnection()
        d.addCallback(writeResponse)


class FingerService(service.Service):
    def __init__(self, filename):
        self.users = {}
        self.filename = filename

    def _read(self):
        with open(self.filename, "rb") as f:
            for line in f:
                user, status = line.split(b':', 1)
                user = user.strip()
                status = status.strip()
                self.users[user] = status
        self.call = reactor.callLater(30, self._read)

    def startService(self):
        self._read()
```

```
        service.Service.startService(self)

    def stopService(self):
        service.Service.stopService(self)
        self.call.cancel()

    def getUser(self, user):
        return defer.succeed(self.users.get(user, b"No such user"))

    def getFingerFactory(self):
        f = protocol.ServerFactory()
        f.protocol = FingerProtocol
        f.getUser = self.getUser
        return f


application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
finger = strports.service("tcp:79", f.getFingerFactory())

finger.setServiceParent(service.IServiceCollection(application))
f.setServiceParent(service.IServiceCollection(application))
```

Since this version is reading data from a file (and refreshing the data every 30 seconds), there is no `FingerSetterFactory` and thus nothing listening on port 1079.

Here we override the standard startService and stopService hooks in the Finger service, which is set up as a child service of the application in the last line of the code. `startService` calls `_read` , the function responsible for reading the data; `reactor.callLater` is then used to schedule it to run again after thirty seconds every time it is called. `reactor.callLater` returns an object that lets us cancel the scheduled run in `stopService` using its `cancel` method.

Announce on Web, Too

The same kind of service can also produce things useful for other protocols. For example, in twisted.web, the factory itself (`Site`) is almost never subclassed — instead, it is given a resource, which represents the tree of resources available via URLs. That hierarchy is navigated by `Site` and overriding it dynamically is possible with `getChild` .

To integrate this into the Finger application (just because we can), we set up a new TCPServer that calls the `Site` factory and retrieves resources via a new function of `FingerService` named `getResource` . This function specifically returns a `Resource` object with an overridden `getChild` method.

`finger15.tac`

```
# Read from file, announce on the web!
from twisted.application import service, strports
from twisted.internet import protocol, reactor, defer
from twisted.protocols import basic
from twisted.web import resource, server, static
import cgi

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        d = self.factory.getUser(user)

        def onError(err):
            return b'Internal error in server'
```

```
        d.addErrback(onError)

        def writeResponse(message):
            self.transport.write(message + b'\r\n')
            self.transport.loseConnection()
        d.addCallback(writeResponse)


class FingerResource(resource.Resource):

    def __init__(self, users):
        self.users = users
        resource.Resource.__init__(self)

    # we treat the path as the username
    def getChild(self, username, request):
        """
        'username' is L{bytes}.
        'request' is a 'twisted.web.server.Request'.
        """
        messagevalue = self.users.get(username)
        if messagevalue:
            messagevalue = messagevalue.decode("ascii")
        if username:
            username = username.decode("ascii")
        username = cgi.escape(username)
        if messagevalue is not None:
            messagevalue = cgi.escape(messagevalue)
            text = '<h1>{}</h1><p>{}</p>'.format(username, messagevalue)
        else:
            text = '<h1>{}</h1><p>No such user</p>'.format(username)
        text = text.encode("ascii")
        return static.Data(text, 'text/html')


class FingerService(service.Service):
    def __init__(self, filename):
        self.filename = filename
        self.users = {}

    def _read(self):
        self.users.clear()
        with open(self.filename, "rb") as f:
            for line in f:
                user, status = line.split(b':', 1)
                user = user.strip()
                status = status.strip()
                self.users[user] = status
        self.call = reactor.callLater(30, self._read)

    def getUser(self, user):
        return defer.succeed(self.users.get(user, b"No such user"))

    def getFingerFactory(self):
        f = protocol.ServerFactory()
        f.protocol = FingerProtocol
        f.getUser = self.getUser
        return f
```

```
    def getResource(self):
        r = FingerResource(self.users)
        return r

    def startService(self):
        self._read()
        service.Service.startService(self)

    def stopService(self):
        service.Service.stopService(self)
        self.call.cancel()


application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
serviceCollection = service.IServiceCollection(application)
f.setServiceParent(serviceCollection)
strports.service("tcp:79", f.getFingerFactory()
                 ).setServiceParent(serviceCollection)
strports.service("tcp:8000", server.Site(f.getResource())
                 ).setServiceParent(serviceCollection)
```

Announce on IRC, Too

This is the first time there is client code. IRC clients often act a lot like servers: responding to events from the network. The Client Service will make sure that severed links will get re-established, with intelligent tweaked exponential back-off algorithms. The IRC client itself is simple: the only real hack is getting the nickname from the factory in `connectionMade`.

`finger16.tac`

```
# Read from file, announce on the web, irc
from twisted.application import internet, service, strports
from twisted.internet import protocol, reactor, defer, endpoints
from twisted.words.protocols import irc
from twisted.protocols import basic
from twisted.web import resource, server, static

import cgi

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        d = self.factory.getUser(user)

        def onError(err):
            return b'Internal error in server'
        d.addErrback(onError)

        def writeResponse(message):
            self.transport.write(message + b'\r\n')
            self.transport.loseConnection()
        d.addCallback(writeResponse)


class IRCReplyBot(irc.IRCClient):
```

```
    def connectionMade(self):
        self.nickname = self.factory.nickname
        irc.IRCClient.connectionMade(self)

    def privmsg(self, user, channel, msg):
        user = user.split('!')[0]
        if self.nickname.lower() == channel.lower():
            d = self.factory.getUser(msg.encode("ascii"))

            def onError(err):
                return b'Internal error in server'
            d.addErrback(onError)

            def writeResponse(message):
                message = message.decode("ascii")
                irc.IRCClient.msg(self, user, msg + ': ' + message)
            d.addCallback(writeResponse)


class FingerService(service.Service):
    def __init__(self, filename):
        self.filename = filename
        self.users = {}

    def _read(self):
        self.users.clear()
        with open(self.filename, "rb") as f:
            for line in f:
                user, status = line.split(b':', 1)
                user = user.strip()
                status = status.strip()
                self.users[user] = status
        self.call = reactor.callLater(30, self._read)

    def getUser(self, user):
        return defer.succeed(self.users.get(user, b"No such user"))

    def getFingerFactory(self):
        f = protocol.ServerFactory()
        f.protocol = FingerProtocol
        f.getUser = self.getUser
        return f

    def getResource(self):
        def getData(path, request):
            user = self.users.get(path, b"No such users <p/> usage: site/user")
            path = path.decode("ascii")
            user = user.decode("ascii")
            text = '<h1>{}</h1><p>{}</p>'.format(path, user)
            text = text.encode("ascii")
            return static.Data(text, 'text/html')

        r = resource.Resource()
        r.getChild = getData
        return r

    def getIRCBot(self, nickname):
        f = protocol.ClientFactory()
```

```
        f.protocol = IRCReplyBot
        f.nickname = nickname
        f.getUser = self.getUser
        return f

    def startService(self):
        self._read()
        service.Service.startService(self)

    def stopService(self):
        service.Service.stopService(self)
        self.call.cancel()


application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
serviceCollection = service.IServiceCollection(application)
f.setServiceParent(serviceCollection)
strports.service("tcp:79", f.getFingerFactory()
                 ).setServiceParent(serviceCollection)
strports.service("tcp:8000", server.Site(f.getResource())
                 ).setServiceParent(serviceCollection)
internet.ClientService(
    endpoints.clientFromString(reactor, "tcp:irc.freenode.org:6667"),
    f.getIRCBot('fingerbot')).setServiceParent(serviceCollection)
```

`FingerService` now has another new function, `getIRCbot` , which returns a `ClientFactory` . This factory in turn will instantiate the `IRCReplyBot` protocol. The IRCBot is configured in the last line to connect to `irc.freenode.org` with a nickname of `fingerbot` .

By overriding `irc.IRCClient.connectionMade` , `IRCReplyBot` can access the `nickname` attribute of the factory that instantiated it.

Add XML-RPC Support

In Twisted, XML-RPC support is handled just as though it was another resource. That resource will still support GET calls normally through render(), but that is usually left unimplemented. Note that it is possible to return deferreds from XML-RPC methods. The client, of course, will not get the answer until the deferred is triggered.

`finger17.tac`

```
# Read from file, announce on the web, irc, xml-rpc
from twisted.application import internet, service, strports
from twisted.internet import protocol, reactor, defer, endpoints
from twisted.words.protocols import irc
from twisted.protocols import basic
from twisted.web import resource, server, static, xmlrpc
import cgi

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        d = self.factory.getUser(user)

        def onError(err):
            return b'Internal error in server'
        d.addErrback(onError)
```

```
        def writeResponse(message):
            self.transport.write(message + b'\r\n')
            self.transport.loseConnection()
        d.addCallback(writeResponse)


class IRCReplyBot(irc.IRCClient):
    def connectionMade(self):
        self.nickname = self.factory.nickname
        irc.IRCClient.connectionMade(self)

    def privmsg(self, user, channel, msg):
        user = user.split('!')[0]
        if self.nickname.lower() == channel.lower():
            d = self.factory.getUser(msg.encode("ascii"))

            def onError(err):
                return 'Internal error in server'
            d.addErrback(onError)

            def writeResponse(message):
                message = message.decode("ascii")
                irc.IRCClient.msg(self, user, msg+ ': ' + message)
            d.addCallback(writeResponse)


class FingerService(service.Service):
    def __init__(self, filename):
        self.filename = filename
        self.users = {}

    def _read(self):
        self.users.clear()
        with open(self.filename, "rb") as f:
            for line in f:
                user, status = line.split(b':', 1)
                user = user.strip()
                status = status.strip()
                self.users[user] = status
        self.call = reactor.callLater(30, self._read)

    def getUser(self, user):
        return defer.succeed(self.users.get(user, b"No such user"))

    def getFingerFactory(self):
        f = protocol.ServerFactory()
        f.protocol = FingerProtocol
        f.getUser = self.getUser
        return f

    def getResource(self):
        def getData(path, request):
            user = self.users.get(path, b"No such user <p/> usage: site/user")
            path = path.decode("ascii")
            user = user.decode("ascii")
            text = '<h1>{}</h1><p>{}</p>'.format(path, user)
            text = text.encode("ascii")
            return static.Data(text, 'text/html')
```

```
        r = resource.Resource()
        r.getChild = getData
        x = xmlrpc.XMLRPC()
        x.xmlrpc_getUser = self.getUser
        r.putChild('RPC2', x)
        return r

    def getIRCBot(self, nickname):
        f = protocol.ClientFactory()
        f.protocol = IRCReplyBot
        f.nickname = nickname
        f.getUser = self.getUser
        return f

    def startService(self):
        self._read()
        service.Service.startService(self)

    def stopService(self):
        service.Service.stopService(self)
        self.call.cancel()


application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
serviceCollection = service.IServiceCollection(application)
f.setServiceParent(serviceCollection)
strports.service("tcp:79", f.getFingerFactory()
                 ).setServiceParent(serviceCollection)
strports.service("tcp:8000", server.Site(f.getResource())
                 ).setServiceParent(serviceCollection)
internet.ClientService(
    endpoints.clientFromString(reactor, "tcp:irc.freenode.org:6667"),
    f.getIRCBot('fingerbot')).setServiceParent(serviceCollection)
```

Instead of a web browser, we can test the XMLRPC finger using a simple client based on Python's built-in `xmlrpclib`, which will access the resource we've made available at `localhost/RPC2`.

`fingerXRclient.py`

```
# testing xmlrpc finger

try:
    # Python 3
    from xmlrpc.client import Server
except ImportError:
    # Python 2
    from xmlrpclib import Server

server = Server('http://127.0.0.1:8000/RPC2')
print(server.getUser('moshez'))
```

The Evolution of Finger: cleaning up the finger code

Introduction

This is the third part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* .

In this section of the tutorial, we'll clean up our code so that it is closer to a readable and extensible style.

Write Readable Code

The last version of the application had a lot of hacks. We avoided sub-classing, didn't support things like user listings over the web, and removed all blank lines – all in the interest of code which is shorter. Here we take a step back, subclass what is more naturally a subclass, make things which should take multiple lines take them, etc. This shows a much better style of developing Twisted applications, though the hacks in the previous stages are sometimes used in throw-away prototypes.

`finger18.tac`

```
# Do everything properly
from twisted.application import internet, service, strports
from twisted.internet import protocol, reactor, defer, endpoints
from twisted.words.protocols import irc
from twisted.protocols import basic
from twisted.web import resource, server, static, xmlrpc
import cgi

def catchError(err):
    return "Internal error in server"


class FingerProtocol(basic.LineReceiver):

    def lineReceived(self, user):
        d = self.factory.getUser(user)
        d.addErrback(catchError)
        def writeValue(value):
            self.transport.write(value + b'\r\n')
            self.transport.loseConnection()
        d.addCallback(writeValue)


class IRCReplyBot(irc.IRCClient):

    def connectionMade(self):
        self.nickname = self.factory.nickname
        irc.IRCClient.connectionMade(self)

    def privmsg(self, user, channel, msg):
        user = user.split('!')[0]
        if self.nickname.lower() == channel.lower():
            d = self.factory.getUser(msg.encode("ascii"))
            d.addErrback(catchError)
            d.addCallback(lambda m: "Status of %s: %s" % (msg, m))
            d.addCallback(lambda m: self.msg(user, m))


class UserStatusTree(resource.Resource):
    def __init__(self, service):
        resource.Resource.__init__(self)
```

```
        self.service = service

    def render_GET(self, request):
        d = self.service.getUsers()
        def formatUsers(users):
            l = ['<li><a href="%s">%s</a></li>' % (user, user)
                 for user in users]
            return '<ul>'+''.join(l)+'</ul>'
        d.addCallback(formatUsers)
        d.addCallback(request.write)
        d.addCallback(lambda _: request.finish())
        return server.NOT_DONE_YET

    def getChild(self, path, request):
        if path=="":
            return UserStatusTree(self.service)
        else:
            return UserStatus(path, self.service)


class UserStatus(resource.Resource):

    def __init__(self, user, service):
        resource.Resource.__init__(self)
        self.user = user
        self.service = service

    def render_GET(self, request):
        d = self.service.getUser(self.user)
        d.addCallback(cgi.escape)
        d.addCallback(lambda m:
                      '<h1>%s</h1>'%self.user+'<p>%s</p>'%m)
        d.addCallback(request.write)
        d.addCallback(lambda _: request.finish())
        return server.NOT_DONE_YET


class UserStatusXR(xmlrpc.XMLRPC):

    def __init__(self, service):
        xmlrpc.XMLRPC.__init__(self)
        self.service = service

    def xmlrpc_getUser(self, user):
        return self.service.getUser(user)


class FingerService(service.Service):

    def __init__(self, filename):
        self.filename = filename
        self.users = {}

    def _read(self):
        self.users.clear()
        with open(self.filename, "rb") as f:
            for line in f:
                user, status = line.split(b':', 1)
```

```
                user = user.strip()
                status = status.strip()
                self.users[user] = status
        self.call = reactor.callLater(30, self._read)

    def getUser(self, user):
        return defer.succeed(self.users.get(user, b"No such user"))

    def getUsers(self):
        return defer.succeed(list(self.users.keys()))

    def getFingerFactory(self):
        f = protocol.ServerFactory()
        f.protocol = FingerProtocol
        f.getUser = self.getUser
        return f

    def getResource(self):
        r = UserStatusTree(self)
        x = UserStatusXR(self)
        r.putChild('RPC2', x)
        return r

    def getIRCBot(self, nickname):
        f = protocol.ClientFactory()
        f.protocol = IRCReplyBot
        f.nickname = nickname
        f.getUser = self.getUser
        return f

    def startService(self):
        self._read()
        service.Service.startService(self)

    def stopService(self):
        service.Service.stopService(self)
        self.call.cancel()


application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
serviceCollection = service.IServiceCollection(application)
f.setServiceParent(serviceCollection)
strports.service("tcp:79", f.getFingerFactory()
                 ).setServiceParent(serviceCollection)
strports.service("tcp:8000", server.Site(f.getResource())
                 ).setServiceParent(serviceCollection)
internet.ClientService(
    endpoints.clientFromString(reactor, "tcp:irc.freenode.org:6667"),
    f.getIRCBot('fingerbot')).setServiceParent(serviceCollection)
```

The Evolution of Finger: moving to a component based architecture

Introduction

This is the fourth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* .

In this section of the tutorial, we'll move our code to a component architecture so that adding new features is trivial. See *Interfaces and Adapters* for a more complete discussion of components.

Write Maintainable Code

In the last version, the service class was three times longer than any other class, and was hard to understand. This was because it turned out to have multiple responsibilities. It had to know how to access user information, by rereading the file every half minute, but also how to display itself in a myriad of protocols. Here, we used the component-based architecture that Twisted provides to achieve a separation of concerns. All the service is responsible for, now, is supporting `getUser` /`getUsers` . It declares its support via the `zope.interface.implementer` decorator. Then, adapters are used to make this service look like an appropriate class for various things: for supplying a finger factory to `TCPServer` , for supplying a resource to site's constructor, and to provide an IRC client factory for `TCPClient` . All the adapters use are the methods in `FingerService` they are declared to use:`getUser` /`getUsers` . We could, of course, skip the interfaces and let the configuration code use things like `FingerFactoryFromService(f)` directly. However, using interfaces provides the same flexibility inheritance gives: future subclasses can override the adapters.

`finger19.tac`

```
# Do everything properly, and componentize
from twisted.application import internet, service, strports
from twisted.internet import protocol, reactor, defer, endpoints
from twisted.words.protocols import irc
from twisted.protocols import basic
from twisted.python import components
from twisted.web import resource, server, static, xmlrpc
from zope.interface import Interface, implementer
import cgi

class IFingerService(Interface):

    def getUser(user):
        """
        Return a deferred returning L{bytes}.
        """

    def getUsers():
        """
        Return a deferred returning a L{list} of L{bytes}.
        """


class IFingerSetterService(Interface):

    def setUser(user, status):
        """
        Set the user's status to something.
        """


def catchError(err):
    return b"Internal error in server"


class FingerProtocol(basic.LineReceiver):

    def lineReceived(self, user):
```

```
        d = self.factory.getUser(user)
        d.addErrback(catchError)
        def writeValue(value):
            self.transport.write(value + b'\r\n')
            self.transport.loseConnection()
        d.addCallback(writeValue)


class IFingerFactory(Interface):

    def getUser(user):
        """
        Return a deferred returning L{bytes}
        """

    def buildProtocol(addr):
        """
        Return a protocol returning L{bytes}
        """


@implementer(IFingerFactory)
class FingerFactoryFromService(protocol.ServerFactory):

    protocol = FingerProtocol

    def __init__(self, service):
        self.service = service

    def getUser(self, user):
        return self.service.getUser(user)

components.registerAdapter(FingerFactoryFromService,
                           IFingerService,
                           IFingerFactory)


class FingerSetterProtocol(basic.LineReceiver):

    def connectionMade(self):
        self.lines = []

    def lineReceived(self, line):
        self.lines.append(line)

    def connectionLost(self, reason):
        if len(self.lines) == 2:
            self.factory.setUser(*self.lines)


class IFingerSetterFactory(Interface):

    def setUser(user, status):
        """
        Return a deferred returning L{bytes}.
        """

    def buildProtocol(addr):
```

```
        """
        Return a protocol returning L{bytes}.
        """


@implementer(IFingerSetterFactory)
class FingerSetterFactoryFromService(protocol.ServerFactory):

    protocol = FingerSetterProtocol

    def __init__(self, service):
        self.service = service

    def setUser(self, user, status):
        self.service.setUser(user, status)


components.registerAdapter(FingerSetterFactoryFromService,
                           IFingerSetterService,
                           IFingerSetterFactory)


class IRCReplyBot(irc.IRCClient):

    def connectionMade(self):
        self.nickname = self.factory.nickname
        irc.IRCClient.connectionMade(self)

    def privmsg(self, user, channel, msg):
        user = user.split('!')[0]
        if self.nickname.lower() == channel.lower():
            d = self.factory.getUser(msg.encode("ascii"))
            d.addErrback(catchError)
            d.addCallback(lambda m: "Status of %s: %s" % (msg, m))
            d.addCallback(lambda m: self.msg(user, m))


class IIRCClientFactory(Interface):
    """
    @ivar nickname
    """

    def getUser(user):
        """
        Return a deferred returning a string.
        """

    def buildProtocol(addr):
        """
        Return a protocol.
        """


@implementer(IIRCClientFactory)
class IRCClientFactoryFromService(protocol.ClientFactory):
    protocol = IRCReplyBot
    nickname = None
```

```
    def __init__(self, service):
        self.service = service

    def getUser(self, user):
        return self.service.getUser(user)

components.registerAdapter(IRCClientFactoryFromService,
                           IFingerService,
                           IIRCClientFactory)


@implementer(resource.IResource)
class UserStatusTree(resource.Resource):
    def __init__(self, service):
        resource.Resource.__init__(self)
        self.service = service
        self.putChild('RPC2', UserStatusXR(self.service))

    def render_GET(self, request):
        d = self.service.getUsers()
        def formatUsers(users):
            l = ['<li><a href="%s">%s</a></li>' % (user, user)
                 for user in users]
            return '<ul>'+''.join(l)+'</ul>'
        d.addCallback(formatUsers)
        d.addCallback(request.write)
        d.addCallback(lambda _: request.finish())
        return server.NOT_DONE_YET

    def getChild(self, path, request):
        if path=="":
            return UserStatusTree(self.service)
        else:
            return UserStatus(path, self.service)

components.registerAdapter(UserStatusTree, IFingerService,
                           resource.IResource)


class UserStatus(resource.Resource):

    def __init__(self, user, service):
        resource.Resource.__init__(self)
        self.user = user
        self.service = service

    def render_GET(self, request):
        d = self.service.getUser(self.user)
        d.addCallback(cgi.escape)
        d.addCallback(lambda m:
                      '<h1>%s</h1>'%self.user+'<p>%s</p>'%m)
        d.addCallback(request.write)
        d.addCallback(lambda _: request.finish())
        return server.NOT_DONE_YET


class UserStatusXR(xmlrpc.XMLRPC):
```

```
    def __init__(self, service):
        xmlrpc.XMLRPC.__init__(self)
        self.service = service

    def xmlrpc_getUser(self, user):
        return self.service.getUser(user)


@implementer(IFingerService)
class FingerService(service.Service):
    def __init__(self, filename):
        self.filename = filename
        self.users = {}

    def _read(self):
        self.users.clear()
        with open(self.filename, "rb") as f:
            for line in f:
                user, status = line.split(b':', 1)
                user = user.strip()
                status = status.strip()
                self.users[user] = status
        self.call = reactor.callLater(30, self._read)

    def getUser(self, user):
        return defer.succeed(self.users.get(user, b"No such user"))

    def getUsers(self):
        return defer.succeed(list(self.users.keys()))

    def startService(self):
        self._read()
        service.Service.startService(self)

    def stopService(self):
        service.Service.stopService(self)
        self.call.cancel()


application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
serviceCollection = service.IServiceCollection(application)
f.setServiceParent(serviceCollection)
strports.service("tcp:79", IFingerFactory(f)
                 ).setServiceParent(serviceCollection)
strports.service("tcp:8000", server.Site(resource.IResource(f))
                 ).setServiceParent(serviceCollection)
i = IIRCClientFactory(f)
i.nickname = 'fingerbot'
internet.ClientService(
    endpoints.clientFromString(reactor, "tcp:irc.freenode.org:6667"),
    i).setServiceParent(serviceCollection)
```

Advantages of Latest Version

- Readable – each class is short

- Maintainable – each class knows only about interfaces
- Dependencies between code parts are minimized
- Example: writing a new `IFingerService` is easy

`finger19a_changes.py`

```
class IFingerSetterService(Interface):

    def setUser(user, status):
        """Set the user's status to something"""

# Advantages of latest version

@implementer(IFingerService, IFingerSetterService)
class MemoryFingerService(service.Service):
    def __init__(self, users):
        self.users = users

    def getUser(self, user):
        return defer.succeed(self.users.get(user, b"No such user"))

    def getUsers(self):
        return defer.succeed(list(self.users.keys()))

    def setUser(self, user, status):
        self.users[user] = status


f = MemoryFingerService({b'moshez': b'Happy and well'})
serviceCollection = service.IServiceCollection(application)
strports.service("tcp:1079:interface=127.0.0.1", IFingerSetterFactory(f)
                 ).setServiceParent(serviceCollection)
```

Full source code here:

`finger19a.tac`

```
# Do everything properly, and componentize
from twisted.application import internet, service, strports
from twisted.internet import protocol, reactor, defer, endpoints
from twisted.words.protocols import irc
from twisted.protocols import basic
from twisted.python import components
from twisted.web import resource, server, static, xmlrpc
from zope.interface import Interface, implementer
import cgi

class IFingerService(Interface):

    def getUser(user):
        """Return a deferred returning L{bytes}"""

    def getUsers():
        """Return a deferred returning a L{list} of L{bytes}"""

class IFingerSetterService(Interface):
```

```
    def setUser(user, status):
        """Set the user's status to something"""

def catchError(err):
    return "Internal error in server"

class FingerProtocol(basic.LineReceiver):

    def lineReceived(self, user):
        d = self.factory.getUser(user)
        d.addErrback(catchError)
        def writeValue(value):
            self.transport.write(value + b'\r\n')
            self.transport.loseConnection()
        d.addCallback(writeValue)


class IFingerFactory(Interface):

    def getUser(user):
        """Return a deferred returning L{bytes}"""

    def buildProtocol(addr):
        """Return a protocol returning L{bytes}"""


@implementer(IFingerFactory)
class FingerFactoryFromService(protocol.ServerFactory):
    protocol = FingerProtocol

    def __init__(self, service):
        self.service = service

    def getUser(self, user):
        return self.service.getUser(user)

components.registerAdapter(FingerFactoryFromService,
                           IFingerService,
                           IFingerFactory)

class FingerSetterProtocol(basic.LineReceiver):

    def connectionMade(self):
        self.lines = []

    def lineReceived(self, line):
        self.lines.append(line)

    def connectionLost(self, reason):
        if len(self.lines) == 2:
            self.factory.setUser(*self.lines)


class IFingerSetterFactory(Interface):

    def setUser(user, status):
        """Return a deferred returning L{bytes}"""
```

```
    def buildProtocol(addr):
        """Return a protocol returning L{bytes}"""


@implementer(IFingerSetterFactory)
class FingerSetterFactoryFromService(protocol.ServerFactory):
    protocol = FingerSetterProtocol

    def __init__(self, service):
        self.service = service

    def setUser(self, user, status):
        self.service.setUser(user, status)


components.registerAdapter(FingerSetterFactoryFromService,
                           IFingerSetterService,
                           IFingerSetterFactory)

class IRCReplyBot(irc.IRCClient):

    def connectionMade(self):
        self.nickname = self.factory.nickname
        irc.IRCClient.connectionMade(self)

    def privmsg(self, user, channel, msg):
        user = user.split('!')[0]
        if self.nickname.lower() == channel.lower():
            d = self.factory.getUser(msg)
            d.addErrback(catchError)
            d.addCallback(lambda m: "Status of %s: %s" % (msg, m))
            d.addCallback(lambda m: self.msg(user, m))


class IIRCClientFactory(Interface):

    """
    @ivar nickname
    """

    def getUser(user):
        """Return a deferred returning a string"""

    def buildProtocol(addr):
        """Return a protocol"""


@implementer(IIRCClientFactory)
class IRCClientFactoryFromService(protocol.ClientFactory):
    protocol = IRCReplyBot
    nickname = None

    def __init__(self, service):
        self.service = service

    def getUser(self, user):
        return self.service.getUser(user)
```

```
components.registerAdapter(IRCClientFactoryFromService,
                           IFingerService,
                           IIRCClientFactory)

@implementer(resource.IResource)
class UserStatusTree(resource.Resource):
    def __init__(self, service):
        resource.Resource.__init__(self)
        self.service = service
        self.putChild('RPC2', UserStatusXR(self.service))

    def render_GET(self, request):
        d = self.service.getUsers()
        def formatUsers(users):
            l = ['<li><a href="%s">%s</a></li>' % (user, user)
                 for user in users]
            return '<ul>'+''.join(l)+'</ul>'
        d.addCallback(formatUsers)
        d.addCallback(request.write)
        d.addCallback(lambda _: request.finish())
        return server.NOT_DONE_YET

    def getChild(self, path, request):
        if path=="":
            return UserStatusTree(self.service)
        else:
            return UserStatus(path, self.service)

components.registerAdapter(UserStatusTree, IFingerService,
                           resource.IResource)

class UserStatus(resource.Resource):

    def __init__(self, user, service):
        resource.Resource.__init__(self)
        self.user = user
        self.service = service

    def render_GET(self, request):
        d = self.service.getUser(self.user)
        d.addCallback(cgi.escape)
        d.addCallback(lambda m:
                      '<h1>%s</h1>'%self.user+'<p>%s</p>'%m)
        d.addCallback(request.write)
        d.addCallback(lambda _: request.finish())
        return server.NOT_DONE_YET


class UserStatusXR(xmlrpc.XMLRPC):

    def __init__(self, service):
        xmlrpc.XMLRPC.__init__(self)
        self.service = service

    def xmlrpc_getUser(self, user):
        return self.service.getUser(user)

@implementer(IFingerService, IFingerSetterService)
```

```
class MemoryFingerService(service.Service):
    def __init__(self, users):
        self.users = users

    def getUser(self, user):
        return defer.succeed(self.users.get(user, b"No such user"))

    def getUsers(self):
        return defer.succeed(list(self.users.keys()))

    def setUser(self, user, status):
        self.users[user] = status


application = service.Application('finger', uid=1, gid=1)
f = MemoryFingerService({b'moshez': b'Happy and well')
serviceCollection = service.IServiceCollection(application)
strports.service("tcp:79", IFingerFactory(f)
                    ).setServiceParent(serviceCollection)
strports.service("tcp:8000", server.Site(resource.IResource(f))
                    ).setServiceParent(serviceCollection)
i = IIRCClientFactory(f)
i.nickname = 'fingerbot'
internet.ClientService(
    endpoints.clientFromString(reactor, "tcp:irc.freenode.org:6667"),
    i).setServiceParent(serviceCollection)
strports.service("tcp:1079:interface=127.0.0.1", IFingerSetterFactory(f)
                    ).setServiceParent(serviceCollection)
```

Aspect-Oriented Programming

At last, an example of aspect-oriented programming that isn't about logging or timing. This code is actually useful! Watch how aspect-oriented programming helps you write less code and have fewer dependencies!

The Evolution of Finger: pluggable backends

Introduction

This is the fifth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* .

In this part we will add new several new backends to our finger service using the component-based architecture developed in *The Evolution of Finger: moving to a component based architecture* . This will show just how convenient it is to implement new back-ends when we move to a component based architecture. Note that here we also use an interface we previously wrote, `FingerSetterFactory` , by supporting one single method. We manage to preserve the service's ignorance of the network.

Another Back-end

`finger19b_changes.py`

```
from twisted.internet import protocol, reactor, defer, utils
import pwd
```

```
# Another back-end

@implementer(IFingerService)
class LocalFingerService(service.Service):
    def getUser(self, user):
    # need a local finger daemon running for this to work
        return utils.getProcessOutput("finger", [user])

    def getUsers(self):
        return defer.succeed([])


f = LocalFingerService()
```

Full source code here:

`finger19b.tac`

```
# Do everything properly, and componentize
from twisted.application import internet, service, strports
from twisted.internet import protocol, reactor, defer, utils, endpoints
from twisted.words.protocols import irc
from twisted.protocols import basic
from twisted.python import components
from twisted.web import resource, server, static, xmlrpc
from zope.interface import Interface, implementer
import cgi
import pwd

class IFingerService(Interface):

    def getUser(user):
        """
        Return a deferred returning L{bytes}.
        """

    def getUsers():
        """
        Return a deferred returning a L{list} of L{bytes}.
        """


class IFingerSetterService(Interface):

    def setUser(user, status):
        """
        Set the user's status to something.
        """


class IFingerSetterService(Interface):

    def setUser(user, status):
        """
        Set the user's status to something.
        """
```

```
def catchError(err):
    return "Internal error in server"


class FingerProtocol(basic.LineReceiver):

    def lineReceived(self, user):
        d = self.factory.getUser(user)
        d.addErrback(catchError)
        def writeValue(value):
            self.transport.write(value + b'\r\n')
            self.transport.loseConnection()
        d.addCallback(writeValue)


class IFingerFactory(Interface):

    def getUser(user):
        """
        Return a deferred returning a string.
        """

    def buildProtocol(addr):
        """
        Return a protocol returning a string.
        """


@implementer(IFingerFactory)
class FingerFactoryFromService(protocol.ServerFactory):
    protocol = FingerProtocol

    def __init__(self, service):
        self.service = service

    def getUser(self, user):
        return self.service.getUser(user)

components.registerAdapter(FingerFactoryFromService,
                           IFingerService,
                           IFingerFactory)


class FingerSetterProtocol(basic.LineReceiver):

    def connectionMade(self):
        self.lines = []

    def lineReceived(self, line):
        self.lines.append(line)

    def connectionLost(self, reason):
        if len(self.lines) == 2:
            self.factory.setUser(*self.lines)


class IFingerSetterFactory(Interface):
```

```
    def setUser(user, status):
        """
        Return a deferred returning L{bytes}.
        """

    def buildProtocol(addr):
        """
        Return a protocol returning L{bytes}.
        """


@implementer(IFingerSetterFactory)
class FingerSetterFactoryFromService(protocol.ServerFactory):
    protocol = FingerSetterProtocol

    def __init__(self, service):
        self.service = service

    def setUser(self, user, status):
        self.service.setUser(user, status)


components.registerAdapter(FingerSetterFactoryFromService,
                           IFingerSetterService,
                           IFingerSetterFactory)

class IRCReplyBot(irc.IRCClient):

    def connectionMade(self):
        self.nickname = self.factory.nickname
        irc.IRCClient.connectionMade(self)

    def privmsg(self, user, channel, msg):
        user = user.split('!')[0]
        if self.nickname.lower() == channel.lower():
            d = self.factory.getUser(msg)
            d.addErrback(catchError)
            d.addCallback(lambda m: "Status of %s: %s" % (msg, m))
            d.addCallback(lambda m: self.msg(user, m))


class IIRCClientFactory(Interface):

    """
    @ivar nickname
    """

    def getUser(user):
        """
        Return a deferred returning L{bytes}.
        """

    def buildProtocol(addr):
        """
        Return a protocol.
        """
```

```
@implementer(IIRCClientFactory)
class IRCClientFactoryFromService(protocol.ClientFactory):
    protocol = IRCReplyBot
    nickname = None

    def __init__(self, service):
        self.service = service

    def getUser(self, user):
        return self.service.getUser(user)

components.registerAdapter(IRCClientFactoryFromService,
                           IFingerService,
                           IIRCClientFactory)


@implementer(resource.IResource)
class UserStatusTree(resource.Resource):
    def __init__(self, service):
        resource.Resource.__init__(self)
        self.service = service
        self.putChild('RPC2', UserStatusXR(self.service))

    def render_GET(self, request):
        d = self.service.getUsers()
        def formatUsers(users):
            l = ['<li><a href="%s">%s</a></li>' % (user, user)
                 for user in users]
            return '<ul>'+''.join(l)+'</ul>'
        d.addCallback(formatUsers)
        d.addCallback(request.write)
        d.addCallback(lambda _: request.finish())
        return server.NOT_DONE_YET

    def getChild(self, path, request):
        if path=="":
            return UserStatusTree(self.service)
        else:
            return UserStatus(path, self.service)

components.registerAdapter(UserStatusTree, IFingerService,
                           resource.IResource)


class UserStatus(resource.Resource):

    def __init__(self, user, service):
        resource.Resource.__init__(self)
        self.user = user
        self.service = service

    def render_GET(self, request):
        d = self.service.getUser(self.user)
        d.addCallback(cgi.escape)
        d.addCallback(lambda m:
                      '<h1>%s</h1>'%self.user+'<p>%s</p>'%m)
        d.addCallback(request.write)
```

```
        d.addCallback(lambda _: request.finish())
        return server.NOT_DONE_YET


class UserStatusXR(xmlrpc.XMLRPC):

    def __init__(self, service):
        xmlrpc.XMLRPC.__init__(self)
        self.service = service

    def xmlrpc_getUser(self, user):
        return self.service.getUser(user)


@implementer(IFingerService)
class FingerService(service.Service):
    def __init__(self, filename):
        self.filename = filename
        self.users = {}

    def _read(self):
        self.users.clear()
        with open(self.filename, "rb") as f:
            for line in f:
                user, status = line.split(b':', 1)
                user = user.strip()
                status = status.strip()
                self.users[user] = status
        self.call = reactor.callLater(30, self._read)

    def getUser(self, user):
        return defer.succeed(self.users.get(user, b"No such user"))

    def getUsers(self):
        return defer.succeed(list(self.users.keys()))

    def startService(self):
        self._read()
        service.Service.startService(self)

    def stopService(self):
        service.Service.stopService(self)
        self.call.cancel()


# Another back-end

@implementer(IFingerService)
class LocalFingerService(service.Service):
    def getUser(self, user):
    # need a local finger daemon running for this to work
        return utils.getProcessOutput(b"finger", [user])

    def getUsers(self):
        return defer.succeed([])


application = service.Application('finger', uid=1, gid=1)
```

```
f = LocalFingerService()
serviceCollection = service.IServiceCollection(application)
strports.service("tcp:79", IFingerFactory(f)
                 ).setServiceParent(serviceCollection)
strports.service("tcp:8000", server.Site(resource.IResource(f))
                 ).setServiceParent(serviceCollection)
i = IIRCClientFactory(f)
i.nickname = 'fingerbot'
internet.ClientService(
    endpoints.clientFromString(reactor, "tcp:irc.freenode.org:6667"),
    i).setServiceParent(serviceCollection)
```

We've already written this, but now we get more for less work: the network code is completely separate from the back-end.

Yet Another Back-end: Doing the Standard Thing

finger19c_changes.py

```
from twisted.internet import protocol, reactor, defer, utils
import pwd
import os


# Yet another back-end

@implementer(IFingerService)
class LocalFingerService(service.Service):
    def getUser(self, user):
        user = user.strip()
        try:
            entry = pwd.getpwnam(user)
        except KeyError:
            return defer.succeed("No such user")
        try:
            f = open(os.path.join(entry[5],'.plan'))
        except (IOError, OSError):
            return defer.succeed("No such user")
        with f:
            data = f.read()
        data = data.strip()
        return defer.succeed(data)

    def getUsers(self):
        return defer.succeed([])



f = LocalFingerService()
```

Full source code here:

finger19c.tac

```
# Do everything properly, and componentize
from twisted.application import internet, service, strports
from twisted.internet import protocol, reactor, defer, utils, endpoints
```

```
from twisted.words.protocols import irc
from twisted.protocols import basic
from twisted.python import components
from twisted.web import resource, server, static, xmlrpc
from zope.interface import Interface, implementer
import cgi
import pwd
import os

class IFingerService(Interface):

    def getUser(user):
        """
        Return a deferred returning L{bytes}.
        """

    def getUsers():
        """
        Return a deferred returning a L{list} of L{bytes}.
        """


class IFingerSetterService(Interface):

    def setUser(user, status):
        """
        Set the user's status to something.
        """


class IFingerSetterService(Interface):

    def setUser(user, status):
        """
        Set the user's status to something.
        """


def catchError(err):
    return "Internal error in server"


class FingerProtocol(basic.LineReceiver):

    def lineReceived(self, user):
        d = self.factory.getUser(user)
        d.addErrback(catchError)
        def writeValue(value):
            self.transport.write(value + b'\r\n')
            self.transport.loseConnection()
        d.addCallback(writeValue)


class IFingerFactory(Interface):

    def getUser(user):
        """
        Return a deferred returning a string.
```

```
        """

    def buildProtocol(addr):
        """
        Return a protocol returning a string.
        """


@implementer(IFingerFactory)
class FingerFactoryFromService(protocol.ServerFactory):

    protocol = FingerProtocol

    def __init__(self, service):
        self.service = service

    def getUser(self, user):
        return self.service.getUser(user)

components.registerAdapter(FingerFactoryFromService,
                           IFingerService,
                           IFingerFactory)


class FingerSetterProtocol(basic.LineReceiver):

    def connectionMade(self):
        self.lines = []

    def lineReceived(self, line):
        self.lines.append(line)

    def connectionLost(self, reason):
        if len(self.lines) == 2:
            self.factory.setUser(*self.lines)


class IFingerSetterFactory(Interface):

    def setUser(user, status):
        """
        Return a deferred returning a string.
        """

    def buildProtocol(addr):
        """
        Return a protocol returning a string.
        """


@implementer(IFingerSetterFactory)
class FingerSetterFactoryFromService(protocol.ServerFactory):

    protocol = FingerSetterProtocol

    def __init__(self, service):
        self.service = service
```

```
    def setUser(self, user, status):
        self.service.setUser(user, status)


components.registerAdapter(FingerSetterFactoryFromService,
                           IFingerSetterService,
                           IFingerSetterFactory)


class IRCReplyBot(irc.IRCClient):

    def connectionMade():
        self.nickname = self.factory.nickname
        irc.IRCClient.connectionMade(self)

    def privmsg(self, user, channel, msg):
        user = user.split('!')[0]
        if self.nickname.lower() == channel.lower():
            d = self.factory.getUser(msg)
            d.addErrback(catchError)
            d.addCallback(lambda m: "Status of %s: %s" % (msg, m))
            d.addCallback(lambda m: self.msg(user, m))


class IIRCClientFactory(Interface):

    """
    @ivar nickname
    """

    def getUser(user):
        """
        Return a deferred returning a string.
        """

    def buildProtocol(addr):
        """
        Return a protocol.
        """


@implementer(IIRCClientFactory)
class IRCClientFactoryFromService(protocol.ClientFactory):

    protocol = IRCReplyBot
    nickname = None

    def __init__(self, service):
        self.service = service

    def getUser(self, user):
        return self.service.getUser(user)

components.registerAdapter(IRCClientFactoryFromService,
                           IFingerService,
                           IIRCClientFactory)
```

```
@implementer(resource.IResource)
class UserStatusTree(resource.Resource):

    def __init__(self, service):
        resource.Resource.__init__(self)
        self.service = service
        self.putChild('RPC2', UserStatusXR(self.service))

    def render_GET(self, request):
        d = self.service.getUsers()
        def formatUsers(users):
            l = ['<li><a href="%s">%s</a></li>' % (user, user)
                 for user in users]
            return '<ul>'+''.join(l)+'</ul>'
        d.addCallback(formatUsers)
        d.addCallback(request.write)
        d.addCallback(lambda _: request.finish())
        return server.NOT_DONE_YET

    def getChild(self, path, request):
        if path=="":
            return UserStatusTree(self.service)
        else:
            return UserStatus(path, self.service)

components.registerAdapter(UserStatusTree, IFingerService,
                           resource.IResource)


class UserStatus(resource.Resource):

    def __init__(self, user, service):
        resource.Resource.__init__(self)
        self.user = user
        self.service = service

    def render_GET(self, request):
        d = self.service.getUser(self.user)
        d.addCallback(cgi.escape)
        d.addCallback(lambda m:
                      '<h1>%s</h1>'%self.user+'<p>%s</p>'%m)
        d.addCallback(request.write)
        d.addCallback(lambda _: request.finish())
        return server.NOT_DONE_YET


class UserStatusXR(xmlrpc.XMLRPC):

    def __init__(self, service):
        xmlrpc.XMLRPC.__init__(self)
        self.service = service

    def xmlrpc_getUser(self, user):
        return self.service.getUser(user)


@implementer(IFingerService)
class FingerService(service.Service):
```

```
    def __init__(self, filename):
        self.filename = filename
        self.users = {}

    def _read(self):
        self.users.clear()
        with open(self.filename, "rb") as f:
            for line in f:
                user, status = line.split(b':', 1)
                user = user.strip()
                status = status.strip()
                self.users[user] = status
        self.call = reactor.callLater(30, self._read)

    def getUser(self, user):
        return defer.succeed(self.users.get(user, b"No such user"))

    def getUsers(self):
        return defer.succeed(list(self.users.keys()))

    def startService(self):
        self._read()
        service.Service.startService(self)

    def stopService(self):
        service.Service.stopService(self)
        self.call.cancel()


# Yet another back-end

@implementer(IFingerService)
class LocalFingerService(service.Service):

    def getUser(self, user):
        user = user.strip()
        try:
            entry = pwd.getpwnam(user)
        except KeyError:
            return defer.succeed(b"No such user")
        try:
            f = open(os.path.join(entry[5],'.plan'))
        except (IOError, OSError):
            return defer.succeed(b"No such user")
        with f:
            data = f.read()
        data = data.strip()
        return defer.succeed(data)

    def getUsers(self):
        return defer.succeed([])


application = service.Application('finger', uid=1, gid=1)
f = LocalFingerService()
serviceCollection = service.IServiceCollection(application)
strports.service("tcp:79", IFingerFactory(f)
```

```
                    ).setServiceParent(serviceCollection)
strports.service("tcp:8000", server.Site(resource.IResource(f))
                    ).setServiceParent(serviceCollection)
i = IIRCClientFactory(f)
i.nickname = 'fingerbot'
internet.ClientService(
    endpoints.clientFromString(reactor, "tcp:irc.freenode.org:6667"),
    i).setServiceParent(serviceCollection)
```

Not much to say except that now we can be churn out backends like crazy. Feel like doing a back-end for Advogato , for example? Dig out the XML-RPC client support Twisted has, and get to work!

The Evolution of Finger: a web frontend

Introduction

This is the sixth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* .

In this part, we demonstrate adding a web frontend using simple twisted.web.resource.Resource objects: `UserStatusTree` , which will produce a listing of all users at the base URL (`/`) of our site; `UserStatus` , which gives the status of each user at the location `/username` ; and `UserStatusXR` , which exposes an XMLRPC interface to `getUser` and `getUsers` functions at the URL `/RPC2` .

In this example we construct HTML segments manually. If the web interface was less trivial, we would want to use more sophisticated web templating and design our system so that HTML rendering and logic were clearly separated.

`finger20.tac`

```
# Do everything properly, and componentize
from twisted.application import internet, service, strports
from twisted.internet import protocol, reactor, defer, endpoints
from twisted.words.protocols import irc
from twisted.protocols import basic
from twisted.python import components
from twisted.web import resource, server, static, xmlrpc
from zope.interface import Interface, implementer
import cgi

class IFingerService(Interface):

    def getUser(user):
        """
        Return a deferred returning L{bytes}.
        """

    def getUsers():
        """
        Return a deferred returning a L{list} of L{bytes}.
        """


class IFingerSetterService(Interface):

    def setUser(user, status):
        """
        Set the user's status to something.
        """
```

```
def catchError(err):
    return b"Internal error in server"


class FingerProtocol(basic.LineReceiver):

    def lineReceived(self, user):
        d = self.factory.getUser(user)
        d.addErrback(catchError)
        def writeValue(value):
            self.transport.write(value + b'\r\n')
            self.transport.loseConnection()
        d.addCallback(writeValue)


class IFingerFactory(Interface):

    def getUser(user):
        """
        Return a deferred returning a string.
        """

    def buildProtocol(addr):
        """
        Return a protocol returning a string.
        """


@implementer(IFingerFactory)
class FingerFactoryFromService(protocol.ServerFactory):

    protocol = FingerProtocol

    def __init__(self, service):
        self.service = service

    def getUser(self, user):
        return self.service.getUser(user)

components.registerAdapter(FingerFactoryFromService,
                           IFingerService,
                           IFingerFactory)


class FingerSetterProtocol(basic.LineReceiver):

    def connectionMade(self):
        self.lines = []

    def lineReceived(self, line):
        self.lines.append(line)

    def connectionLost(self, reason):
        if len(self.lines) == 2:
            self.factory.setUser(*self.lines)
```

```
class IFingerSetterFactory(Interface):

    def setUser(user, status):
        """
        Return a deferred returning a string.
        """

    def buildProtocol(addr):
        """
        Return a protocol returning a string.
        """


@implementer(IFingerSetterFactory)
class FingerSetterFactoryFromService(protocol.ServerFactory):

    protocol = FingerSetterProtocol

    def __init__(self, service):
        self.service = service

    def setUser(self, user, status):
        self.service.setUser(user, status)


components.registerAdapter(FingerSetterFactoryFromService,
                           IFingerSetterService,
                           IFingerSetterFactory)


class IRCReplyBot(irc.IRCClient):

    def connectionMade(self):
        self.nickname = self.factory.nickname
        irc.IRCClient.connectionMade(self)

    def privmsg(self, user, channel, msg):
        user = user.split('!')[0]
        if self.nickname.lower() == channel.lower():
            d = self.factory.getUser(msg)
            d.addErrback(catchError)
            d.addCallback(lambda m: "Status of %s: %s" % (msg, m))
            d.addCallback(lambda m: self.msg(user, m))


class IIRCClientFactory(Interface):

    """
    @ivar nickname
    """

    def getUser(user):
        """
        Return a deferred returning a string.
        """

    def buildProtocol(addr):
```

```
        """
        Return a protocol.
        """


@implementer(IIRCClientFactory)
class IRCClientFactoryFromService(protocol.ClientFactory):

    protocol = IRCReplyBot
    nickname = None

    def __init__(self, service):
        self.service = service

    def getUser(self, user):
        return self.service.getUser(user)

components.registerAdapter(IRCClientFactoryFromService,
                           IFingerService,
                           IIRCClientFactory)


class UserStatusTree(resource.Resource):

    def __init__(self, service):
        resource.Resource.__init__(self)
        self.service=service

        # add a specific child for the path "RPC2"
        self.putChild("RPC2", UserStatusXR(self.service))

        # need to do this for resources at the root of the site
        self.putChild("", self)

    def _cb_render_GET(self, users, request):
        userOutput = ''.join(["<li><a href=\"%s\">%s</a></li>" % (user, user)
                for user in users])
        request.write("""
            <html><head><title>Users</title></head><body>
            <h1>Users</h1>
            <ul>
            %s
            </ul></body></html>""" % userOutput)
        request.finish()

    def render_GET(self, request):
        d = self.service.getUsers()
        d.addCallback(self._cb_render_GET, request)

        # signal that the rendering is not complete
        return server.NOT_DONE_YET

    def getChild(self, path, request):
        return UserStatus(user=path, service=self.service)

components.registerAdapter(UserStatusTree, IFingerService, resource.IResource)
```

```
class UserStatus(resource.Resource):

    def __init__(self, user, service):
        resource.Resource.__init__(self)
        self.user = user
        self.service = service

    def _cb_render_GET(self, status, request):
        request.write("""<html><head><title>%s</title></head>
        <body><h1>%s</h1>
        <p>%s</p>
        </body></html>""" % (self.user, self.user, status))
        request.finish()

    def render_GET(self, request):
        d = self.service.getUser(self.user)
        d.addCallback(self._cb_render_GET, request)

        # signal that the rendering is not complete
        return server.NOT_DONE_YET


class UserStatusXR(xmlrpc.XMLRPC):

    def __init__(self, service):
        xmlrpc.XMLRPC.__init__(self)
        self.service = service

    def xmlrpc_getUser(self, user):
        return self.service.getUser(user)

    def xmlrpc_getUsers(self):
        return self.service.getUsers()


@implementer(IFingerService)
class FingerService(service.Service):

    def __init__(self, filename):
        self.filename = filename
        self.users = {}

    def _read(self):
        self.users.clear()
        with open(self.filename, "rb") as f:
            for line in f:
                user, status = line.split(b':', 1)
                user = user.strip()
                status = status.strip()
                self.users[user] = status
        self.call = reactor.callLater(30, self._read)

    def getUser(self, user):
        return defer.succeed(self.users.get(user, b"No such user"))

    def getUsers(self):
        return defer.succeed(list(self.users.keys()))
```

```
    def startService(self):
        self._read()
        service.Service.startService(self)

    def stopService(self):
        service.Service.stopService(self)
        self.call.cancel()


application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
serviceCollection = service.IServiceCollection(application)
f.setServiceParent(serviceCollection)
strports.service("tcp:79", IFingerFactory(f)
                 ).setServiceParent(serviceCollection)
strports.service("tcp:8000", server.Site(resource.IResource(f))
                 ).setServiceParent(serviceCollection)
i = IIRCClientFactory(f)
i.nickname = 'fingerbot'
internet.ClientService(
    endpoints.clientFromString(reactor, "tcp:irc.freenode.org:6667"),
    i).setServiceParent(serviceCollection)
```

The Evolution of Finger: Twisted client support using Perspective Broker

Introduction

This is the seventh part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* .

In this part, we add a Perspective Broker service to the finger application so that Twisted clients can access the finger server. Perspective Broker is introduced in depth in its own *section* of the core howto index.

Use Perspective Broker

We add support for perspective broker, Twisted's native remote object protocol. Now, Twisted clients will not have to go through XML-RPCish contortions to get information about users.

`finger21.tac`

```
# Do everything properly, and componentize
from twisted.application import internet, service, strports
from twisted.internet import protocol, reactor, defer, endpoints
from twisted.words.protocols import irc
from twisted.protocols import basic
from twisted.python import components
from twisted.web import resource, server, static, xmlrpc
from twisted.spread import pb
from zope.interface import Interface, implementer
import cgi

class IFingerService(Interface):

    def getUser(user):
        """
        Return a deferred returning L{bytes}.
```

```
        """

    def getUsers():
        """
        Return a deferred returning a L{list} of L{bytes}.
        """


class IFingerSetterService(Interface):

    def setUser(user, status):
        """
        Set the user's status to something.
        """


def catchError(err):
    return "Internal error in server"


class FingerProtocol(basic.LineReceiver):

    def lineReceived(self, user):
        d = self.factory.getUser(user)
        d.addErrback(catchError)
        def writeValue(value):
            self.transport.write(value+'\r\n')
            self.transport.loseConnection()
        d.addCallback(writeValue)


class IFingerFactory(Interface):

    def getUser(user):
        """
        Return a deferred returning a string.
        """

    def buildProtocol(addr):
        """
        Return a protocol returning a string.
        """


@implementer(IFingerFactory)
class FingerFactoryFromService(protocol.ServerFactory):

    protocol = FingerProtocol

    def __init__(self, service):
        self.service = service

    def getUser(self, user):
        return self.service.getUser(user)

components.registerAdapter(FingerFactoryFromService,
                           IFingerService,
                           IFingerFactory)
```

```
class FingerSetterProtocol(basic.LineReceiver):

    def connectionMade(self):
        self.lines = []

    def lineReceived(self, line):
        self.lines.append(line)

    def connectionLost(self, reason):
        if len(self.lines) == 2:
            self.factory.setUser(*self.lines)


class IFingerSetterFactory(Interface):

    def setUser(user, status):
        """
        Return a deferred returning a string.
        """

    def buildProtocol(addr):
        """
        Return a protocol returning a string.
        """


@implementer(IFingerSetterFactory)
class FingerSetterFactoryFromService(protocol.ServerFactory):

    protocol = FingerSetterProtocol

    def __init__(self, service):
        self.service = service

    def setUser(self, user, status):
        self.service.setUser(user, status)


components.registerAdapter(FingerSetterFactoryFromService,
                           IFingerSetterService,
                           IFingerSetterFactory)


class IRCReplyBot(irc.IRCClient):

    def connectionMade(self):
        self.nickname = self.factory.nickname
        irc.IRCClient.connectionMade(self)

    def privmsg(self, user, channel, msg):
        user = user.split('!')[0]
        if self.nickname.lower() == channel.lower():
            d = self.factory.getUser(msg.encode("ascii"))
            d.addErrback(catchError)
            d.addCallback(lambda m: "Status of %s: %s" % (msg, m))
            d.addCallback(lambda m: self.msg(user, m))
```

```
class IIRCClientFactory(Interface):

    """
    @ivar nickname
    """

    def getUser(user):
        """
        Return a deferred returning a string.
        """

    def buildProtocol(addr):
        """
        Return a protocol.
        """


@implementer(IIRCClientFactory)
class IRCClientFactoryFromService(protocol.ClientFactory):

    protocol = IRCReplyBot
    nickname = None

    def __init__(self, service):
        self.service = service

    def getUser(self, user):
        return self.service.getUser(user)

components.registerAdapter(IRCClientFactoryFromService,
                           IFingerService,
                           IIRCClientFactory)


class UserStatusTree(resource.Resource):

    def __init__(self, service):
        resource.Resource.__init__(self)
        self.service=service

        # add a specific child for the path "RPC2"
        self.putChild("RPC2", UserStatusXR(self.service))

        # need to do this for resources at the root of the site
        self.putChild("", self)

    def _cb_render_GET(self, users, request):
        userOutput = ''.join(["<li><a href=\"%s\">%s</a></li>" % (user, user)
                for user in users])
        request.write("""
            <html><head><title>Users</title></head><body>
            <h1>Users</h1>
            <ul>
            %s
            </ul></body></html>""" % userOutput)
        request.finish()
```

```
    def render_GET(self, request):
        d = self.service.getUsers()
        d.addCallback(self._cb_render_GET, request)

        # signal that the rendering is not complete
        return server.NOT_DONE_YET

    def getChild(self, path, request):
        return UserStatus(user=path, service=self.service)

components.registerAdapter(UserStatusTree, IFingerService, resource.IResource)


class UserStatus(resource.Resource):

    def __init__(self, user, service):
        resource.Resource.__init__(self)
        self.user = user
        self.service = service

    def _cb_render_GET(self, status, request):
        request.write("""<html><head><title>%s</title></head>
        <body><h1>%s</h1>
        <p>%s</p>
        </body></html>""" % (self.user, self.user, status))
        request.finish()

    def render_GET(self, request):
        d = self.service.getUser(self.user)
        d.addCallback(self._cb_render_GET, request)

        # signal that the rendering is not complete
        return server.NOT_DONE_YET


class UserStatusXR(xmlrpc.XMLRPC):

    def __init__(self, service):
        xmlrpc.XMLRPC.__init__(self)
        self.service = service

    def xmlrpc_getUser(self, user):
        return self.service.getUser(user)

    def xmlrpc_getUsers(self):
        return self.service.getUsers()


class IPerspectiveFinger(Interface):

    def remote_getUser(username):
        """
        Return a user's status.
        """

    def remote_getUsers():
        """
```

```
        Return a user's status.
        """


@implementer(IPerspectiveFinger)
class PerspectiveFingerFromService(pb.Root):

    def __init__(self, service):
        self.service = service

    def remote_getUser(self, username):
        return self.service.getUser(username)

    def remote_getUsers(self):
        return self.service.getUsers()

components.registerAdapter(PerspectiveFingerFromService,
                           IFingerService,
                           IPerspectiveFinger)


@implementer(IFingerService)
class FingerService(service.Service):

    def __init__(self, filename):
        self.filename = filename
        self.users = {}

    def _read(self):
        self.users.clear()
        with open(self.filename, "rb") as f:
            for line in f:
                user, status = line.split(b':', 1)
                user = user.strip()
                status = status.strip()
                self.users[user] = status
        self.call = reactor.callLater(30, self._read)

    def getUser(self, user):
        return defer.succeed(self.users.get(user, b"No such user"))

    def getUsers(self):
        return defer.succeed(list(self.users.keys()))

    def startService(self):
        self._read()
        service.Service.startService(self)

    def stopService(self):
        service.Service.stopService(self)
        self.call.cancel()


application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
serviceCollection = service.IServiceCollection(application)
f.setServiceParent(serviceCollection)
strports.service("tcp:79", IFingerFactory(f)
```

```
                    ).setServiceParent(serviceCollection)
strports.service("tcp:8000", server.Site(resource.IResource(f))
                    ).setServiceParent(serviceCollection)
i = IIRCClientFactory(f)
i.nickname = 'fingerbot'
internet.ClientService(
    endpoints.clientFromString(reactor, "tcp:irc.freenode.org:6667"),
    i).setServiceParent(serviceCollection)
strports.service("tcp:8889", pb.PBServerFactory(IPerspectiveFinger(f))
                    ).setServiceParent(serviceCollection)
```

A simple client to test the perspective broker finger:

`fingerPBclient.py`

```
# test the PB finger on port 8889
# this code is essentially the same as
# the first example in howto/pb-usage

from __future__ import print_function

from twisted.spread import pb
from twisted.internet import reactor, endpoints

def gotObject(object):
    print("got object:", object)
    object.callRemote("getUser","moshez").addCallback(gotData)
# or
#   object.callRemote("getUsers").addCallback(gotData)

def gotData(data):
    print('server sent:', data)
    reactor.stop()

def gotNoObject(reason):
    print("no object:",reason)
    reactor.stop()

factory = pb.PBClientFactory()
clientEndpoint = endpoints.clientFromString("tcp:127.0.0.1:8889")
clientEndpoint.connect(factory)
factory.getRootObject().addCallbacks(gotObject,gotNoObject)
reactor.run()
```

The Evolution of Finger: using a single factory for multiple protocols

Introduction

This is the eighth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* .

In this part, we add HTTPS support to our web frontend, showing how to have a single factory listen on multiple ports. More information on using SSL in Twisted can be found in the *SSL howto* .

Support HTTPS

All we need to do to code an HTTPS site is just write a context factory (in this case, which loads the certificate from a certain file) and then use the `twisted.internet.endpoints.serverFromString` method to build a SSL endpoint. Note that one factory (in this case, a site) can listen on multiple ports with multiple protocols.

Of course, this endpoint doesn't work without a TLS certificate and a private key. You'll need to create a self-signed cert and key. This will obviously not be trusted by your web browser, so you'll see a warning when you connect. In this case, don't worry: you're not at risk.

To create a certificate and key that can be used by this tutorial, run the following:

```
openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem -days 365
```

finger22.py

```
# Do everything properly, and componentize
from twisted.application import internet, service, strports
from twisted.internet import protocol, reactor, defer, endpoints
from twisted.words.protocols import irc
from twisted.protocols import basic
from twisted.python import components
from twisted.web import resource, server, static, xmlrpc
from twisted.spread import pb
from zope.interface import Interface, implementer
from OpenSSL import SSL
import cgi

class IFingerService(Interface):

    def getUser(user):
        """
        Return a deferred returning L{bytes}.
        """

    def getUsers():
        """
        Return a deferred returning a L{list} of L{bytes}.
        """


class IFingerSetterService(Interface):

    def setUser(user, status):
        """
        Set the user's status to something.
        """


def catchError(err):
    return "Internal error in server"


class FingerProtocol(basic.LineReceiver):

    def lineReceived(self, user):
        d = self.factory.getUser(user)
        d.addErrback(catchError)
```

```
        def writeValue(value):
            self.transport.write(value + b'\r\n')
            self.transport.loseConnection()
        d.addCallback(writeValue)


class IFingerFactory(Interface):

    def getUser(user):
        """
        Return a deferred returning a string.
        """

    def buildProtocol(addr):
        """
        Return a protocol returning a string.
        """


@implementer(IFingerFactory)
class FingerFactoryFromService(protocol.ServerFactory):

    protocol = FingerProtocol

    def __init__(self, service):
        self.service = service

    def getUser(self, user):
        return self.service.getUser(user)

components.registerAdapter(FingerFactoryFromService,
                           IFingerService,
                           IFingerFactory)


class FingerSetterProtocol(basic.LineReceiver):

    def connectionMade(self):
        self.lines = []

    def lineReceived(self, line):
        self.lines.append(line)

    def connectionLost(self, reason):
        if len(self.lines) == 2:
            self.factory.setUser(*self.lines)


class IFingerSetterFactory(Interface):

    def setUser(user, status):
        """
        Return a deferred returning L{bytes}.
        """

    def buildProtocol(addr):
        """
        Return a protocol returning L{bytes}.
```

```
        """


@implementer(IFingerSetterFactory)
class FingerSetterFactoryFromService(protocol.ServerFactory):

    protocol = FingerSetterProtocol

    def __init__(self, service):
        self.service = service

    def setUser(self, user, status):
        self.service.setUser(user, status)


components.registerAdapter(FingerSetterFactoryFromService,
                           IFingerSetterService,
                           IFingerSetterFactory)


class IRCReplyBot(irc.IRCClient):

    def connectionMade(self):
        self.nickname = self.factory.nickname
        irc.IRCClient.connectionMade(self)

    def privmsg(self, user, channel, msg):
        user = user.split('!')[0]
        if self.nickname.lower() == channel.lower():
            d = self.factory.getUser(msg.encode("ascii"))
            d.addErrback(catchError)
            d.addCallback(lambda m: "Status of %s: %s" % (msg, m))
            d.addCallback(lambda m: self.msg(user, m))


class IIRCClientFactory(Interface):

    """
    @ivar nickname
    """

    def getUser(user):
        """
        Return a deferred returning a string.
        """

    def buildProtocol(addr):
        """
        Return a protocol.
        """


@implementer(IIRCClientFactory)
class IRCClientFactoryFromService(protocol.ClientFactory):

    protocol = IRCReplyBot
    nickname = None
```

```
    def __init__(self, service):
        self.service = service

    def getUser(self, user):
        return self.service.getUser(user)

components.registerAdapter(IRCClientFactoryFromService,
                           IFingerService,
                           IIRCClientFactory)


class UserStatusTree(resource.Resource):

    def __init__(self, service):
        resource.Resource.__init__(self)
        self.service=service

        # add a specific child for the path "RPC2"
        self.putChild("RPC2", UserStatusXR(self.service))

        # need to do this for resources at the root of the site
        self.putChild("", self)

    def _cb_render_GET(self, users, request):
        userOutput = ''.join(["<li><a href=\"%s\">%s</a></li>" % (user, user)
                for user in users])
        request.write("""
            <html><head><title>Users</title></head><body>
            <h1>Users</h1>
            <ul>
            %s
            </ul></body></html>""" % userOutput)
        request.finish()

    def render_GET(self, request):
        d = self.service.getUsers()
        d.addCallback(self._cb_render_GET, request)

        # signal that the rendering is not complete
        return server.NOT_DONE_YET

    def getChild(self, path, request):
        return UserStatus(user=path, service=self.service)

components.registerAdapter(UserStatusTree, IFingerService, resource.IResource)


class UserStatus(resource.Resource):

    def __init__(self, user, service):
        resource.Resource.__init__(self)
        self.user = user
        self.service = service

    def _cb_render_GET(self, status, request):
        request.write("""<html><head><title>%s</title></head>
        <body><h1>%s</h1>
        <p>%s</p>
```

```
        </body></html>""" % (self.user, self.user, status))
        request.finish()

    def render_GET(self, request):
        d = self.service.getUser(self.user)
        d.addCallback(self._cb_render_GET, request)

        # signal that the rendering is not complete
        return server.NOT_DONE_YET


class UserStatusXR(xmlrpc.XMLRPC):

    def __init__(self, service):
        xmlrpc.XMLRPC.__init__(self)
        self.service = service

    def xmlrpc_getUser(self, user):
        return self.service.getUser(user)

    def xmlrpc_getUsers(self):
        return self.service.getUsers()


class IPerspectiveFinger(Interface):

    def remote_getUser(username):
        """
        Return a user's status.
        """

    def remote_getUsers():
        """
        Return a user's status.
        """

@implementer(IPerspectiveFinger)
class PerspectiveFingerFromService(pb.Root):

    def __init__(self, service):
        self.service = service

    def remote_getUser(self, username):
        return self.service.getUser(username)

    def remote_getUsers(self):
        return self.service.getUsers()

components.registerAdapter(PerspectiveFingerFromService,
                           IFingerService,
                           IPerspectiveFinger)


@implementer(IFingerService)
class FingerService(service.Service):

    def __init__(self, filename):
        self.filename = filename
```

```
        self.users = {}

    def _read(self):
        self.users.clear()
        with open(self.filename, "rb") as f:
            for line in f:
                user, status = line.split(b':', 1)
                user = user.strip()
                status = status.strip()
                self.users[user] = status
        self.call = reactor.callLater(30, self._read)

    def getUser(self, user):
        return defer.succeed(self.users.get(user, b"No such user"))

    def getUsers(self):
        return defer.succeed(list(self.users.keys()))

    def startService(self):
        self._read()
        service.Service.startService(self)

    def stopService(self):
        service.Service.stopService(self)
        self.call.cancel()



application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
serviceCollection = service.IServiceCollection(application)
f.setServiceParent(serviceCollection)
strports.service("tcp:79", IFingerFactory(f)
                 ).setServiceParent(serviceCollection)
site = server.Site(resource.IResource(f))
strports.service("tcp:8000", site,
                 ).setServiceParent(serviceCollection)
strports.service("ssl:port=443:certKey=cert.pem:privateKey=key.pem", site
                 ).setServiceParent(serviceCollection)
i = IIRCClientFactory(f)
i.nickname = 'fingerbot'
internet.ClientService(
    endpoints.clientFromString(reactor, "tcp:irc.freenode.org:6667"),
    i).setServiceParent(serviceCollection)
strports.service("tcp:8889", pb.PBServerFactory(IPerspectiveFinger(f))
                 ).setServiceParent(serviceCollection)
```

The Evolution of Finger: a Twisted finger client

Introduction

This is the ninth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* .

In this part, we develop a client for the finger server: a proxy finger server which forwards requests to another finger server.

Finger Proxy

Writing new clients with Twisted is much like writing new servers. We implement the protocol, which just gathers up all the data, and give it to the factory. The factory keeps a deferred which is triggered if the connection either fails or succeeds. When we use the client, we first make sure the deferred will never fail, by producing a message in that case. Implementing a wrapper around client which just returns the deferred is a common pattern. While less flexible than using the factory directly, it's also more convenient.

Additionally, because this code now programmatically receives its host and port, it's a bit less convenient to use clientFromString. Instead, we move to using the specific endpoint we want. In this case, because we're connecting as a client over IPv4 using TCP, we want the `TCP4ClientEndpoint`.

`fingerproxy.tac`

```
# finger proxy
from twisted.application import internet, service, strports
from twisted.internet import defer, protocol, reactor, endpoints
from twisted.protocols import basic
from twisted.python import components
from zope.interface import Interface, implementer


def catchError(err):
    return "Internal error in server"

class IFingerService(Interface):

    def getUser(user):
        """Return a deferred returning L{bytes}"""

    def getUsers():
        """Return a deferred returning a L{list} of L{bytes}"""


class IFingerFactory(Interface):

    def getUser(user):
        """Return a deferred returning L{bytes}"""

    def buildProtocol(addr):
        """Return a protocol returning L{bytes}"""

class FingerProtocol(basic.LineReceiver):

    def lineReceived(self, user):
        d = self.factory.getUser(user)
        d.addErrback(catchError)
        def writeValue(value):
            self.transport.write(value)
            self.transport.loseConnection()
        d.addCallback(writeValue)



@implementer(IFingerFactory)
class FingerFactoryFromService(protocol.ClientFactory):

    protocol = FingerProtocol
```

```
    def __init__(self, service):
        self.service = service

    def getUser(self, user):
        return self.service.getUser(user)


components.registerAdapter(FingerFactoryFromService,
                           IFingerService,
                           IFingerFactory)

class FingerClient(protocol.Protocol):

    def connectionMade(self):
        self.transport.write(self.factory.user + b"\r\n")
        self.buf = []

    def dataReceived(self, data):
        self.buf.append(data)

    def connectionLost(self, reason):
        self.factory.gotData(''.join(self.buf))

class FingerClientFactory(protocol.ClientFactory):

    protocol = FingerClient

    def __init__(self, user):
        self.user = user
        self.d = defer.Deferred()

    def clientConnectionFailed(self, _, reason):
        self.d.errback(reason)

    def gotData(self, data):
        self.d.callback(data)


def finger(user, host, port=79):
    f = FingerClientFactory(user)
    endpoint = endpoints.TCP4ClientEndpoint(reactor, host, port)
    endpoint.connect(f)
    return f.d


@implementer(IFingerService)
class ProxyFingerService(service.Service):

    def getUser(self, user):
        try:
            user, host = user.split('@', 1)
        except:
            user = user.strip()
            host = '127.0.0.1'
        ret = finger(user, host)
        ret.addErrback(lambda _: "Could not connect to remote host")
        return ret
```

```
    def getUsers(self):
        return defer.succeed([])

application = service.Application('finger', uid=1, gid=1)
f = ProxyFingerService()
strports.service("tcp:7779", IFingerFactory(f)).setServiceParent(
    service.IServiceCollection(application))
```

The Evolution of Finger: making a finger library

Introduction

This is the tenth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* .

In this part, we separate the application code that launches a finger service from the library code which defines a finger service, placing the application in a Twisted Application Configuration (.tac) file. We also move configuration (such as HTML templates) into separate files. Configuration and deployment with .tac and twistd are introduced in *Using the Twisted Application Framework* .

Organization

Now this code, while quite modular and well-designed, isn't properly organized. Everything above the `application=` belongs in a module, and the HTML templates all belong in separate files.

We can use the `templateFile` and `templateDirectory` attributes to indicate what HTML template file to use for each Page, and where to look for it.

`organized-finger.tac`

```
# organized-finger.tac
# eg:  twistd -ny organized-finger.tac

import finger

from twisted.internet import protocol, reactor, defer, endpoints
from twisted.spread import pb
from twisted.web import resource, server
from twisted.application import internet, service, strports
from twisted.python import log

application = service.Application('finger', uid=1, gid=1)
f = finger.FingerService('/etc/users')
serviceCollection = service.IServiceCollection(application)
strports.service("tcp:79", finger.IFingerFactory(f)
                 ).setServiceParent(serviceCollection)

site = server.Site(resource.IResource(f))
strports.service("tcp:8000", site,
                 ).setServiceParent(serviceCollection)

strports.service("ssl:port=443:certKey=cert.pem:privateKey=key.pem", site
                 ).setServiceParent(serviceCollection)

i = finger.IIRCClientFactory(f)
```

```
i.nickname = 'fingerbot'
internet.ClientService(
    endpoints.clientFromString(reactor, "tcp:irc.freenode.org:6667"),
    i).setServiceParent(serviceCollection)

strports.service("tcp:8889", pb.PBServerFactory(finger.IPerspectiveFinger(f))
                 ).setServiceParent(serviceCollection)
```

Note that our program is now quite separated. We have:

- Code (in the module)
- Configuration (file above)
- Presentation (templates)
- Content (`/etc/users`)
- Deployment (twistd)

Prototypes don't need this level of separation, so our earlier examples all bunched together. However, real applications do. Thankfully, if we write our code correctly, it is easy to achieve a good separation of parts.

Easy Configuration

We can also supply easy configuration for common cases with a `makeService` method that will also help build .tac files later:

`finger_config.py`

```
# Easy configuration
# makeService from finger module

def makeService(config):
    # finger on port 79
    s = service.MultiService()
    f = FingerService(config['file'])
    h = strports.service("tcp:79", IFingerFactory(f))
    h.setServiceParent(s)

    # website on port 8000
    r = resource.IResource(f)
    r.templateDirectory = config['templates']
    site = server.Site(r)
    j = strports.service("tcp:8000", site)
    j.setServiceParent(s)

    # ssl on port 443
    if config.get('ssl'):
        k = strports.service(
            "ssl:port=443:certKey=cert.pem:privateKey=key.pem", site
        )
        k.setServiceParent(s)

    # irc fingerbot
    if 'ircnick' in config:
        i = IIRCClientFactory(f)
        i.nickname = config['ircnick']
        ircserver = config['ircserver']
```

```
        b = internet.ClientService(
            endpoints.HostnameEndpoint(reactor, ircserver, 6667), i
        )
        b.setServiceParent(s)

    # Pespective Broker on port 8889
    if 'pbport' in config:
        m = internet.StreamServerEndpointService(
            endpoints.TCP4ServerEndpoint(reactor, int(config['pbport'])),
            pb.PBServerFactory(IPerspectiveFinger(f))
        )
        m.setServiceParent(s)

    return s
```

And we can write simpler files now:

`simple-finger.tac`

```
# simple-finger.tac
# eg:  twistd -ny simple-finger.tac

from twisted.application import service

import finger

options = { 'file': '/etc/users',
            'templates': '/usr/share/finger/templates',
            'ircnick': 'fingerbot',
            'ircserver': 'irc.freenode.net',
            'pbport': 8889,
            'ssl': 'ssl=0' }

ser = finger.makeService(options)
application = service.Application('finger', uid=1, gid=1)
ser.setServiceParent(service.IServiceCollection(application))
```

```
% twistd -ny simple-finger.tac
```

Note: the finger *user* still has ultimate power: they can use `makeService`, or they can use the lower-level interface if they have specific needs (maybe an IRC server on some other port? Maybe we want the non-SSL webserver to listen only locally? etc. etc.). This is an important design principle: never *force* a layer of abstraction; *allow* usage of layers of abstractions instead.

The pasta theory of design:

Spaghetti Each piece of code interacts with every other piece of code (can be implemented with GOTO, functions, objects).

Lasagna Code has carefully designed layers. Each layer is, in theory independent. However low-level layers usually cannot be used easily, and high-level layers depend on low-level layers.

Ravioli Each part of the code is useful by itself. There is a thin layer of interfaces between various parts (the sauce). Each part can be usefully be used elsewhere.

...but sometimes, the user just wants to order "Ravioli", so one coarse-grain easily definable layer of abstraction on top of it all can be useful.

The Evolution of Finger: configuration of the finger service

Introduction

This is the eleventh part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* .

In this part, we make it easier for non-programmers to configure a finger server. Plugins are discussed further in the *Twisted Plugin System* howto. Writing twistd plugins is covered in *Writing a twistd Plugin*, and .tac applications are covered in *Using the Twisted Application Framework*.

Plugins

So far, the user had to be somewhat of a programmer to be able to configure stuff. Maybe we can eliminate even that? Move old code to `finger/__init__.py` and...

Full source code for finger module here:

`finger.py`

```
# finger.py module

from zope.interface import Interface, implementer

from twisted.application import internet, service, strports
from twisted.internet import protocol, reactor, defer, endpoints
from twisted.words.protocols import irc
from twisted.protocols import basic
from twisted.python import components, log
from twisted.web import resource, server, xmlrpc
from twisted.spread import pb

class IFingerService(Interface):

    def getUser(user):
        """
        Return a deferred returning a L{bytes}.
        """

    def getUsers():
        """
        Return a deferred returning a L{list} of L{bytes}.
        """


class IFingerSetterService(Interface):

    def setUser(user, status):
        """
        Set the user's status to something.
        """


def catchError(err):
    return "Internal error in server"


class FingerProtocol(basic.LineReceiver):
```

```
    def lineReceived(self, user):
        d = self.factory.getUser(user)
        d.addErrback(catchError)
        def writeValue(value):
            self.transport.write(value + b'\n')
            self.transport.loseConnection()
        d.addCallback(writeValue)


class IFingerFactory(Interface):

    def getUser(user):
        """
        Return a deferred returning L{bytes}.
        """

    def buildProtocol(addr):
        """
        Return a protocol returning L{bytes}.
        """


@implementer(IFingerFactory)
class FingerFactoryFromService(protocol.ServerFactory):

    protocol = FingerProtocol

    def __init__(self, service):
        self.service = service

    def getUser(self, user):
        return self.service.getUser(user)

components.registerAdapter(FingerFactoryFromService,
                           IFingerService,
                           IFingerFactory)


class FingerSetterProtocol(basic.LineReceiver):

    def connectionMade(self):
        self.lines = []

    def lineReceived(self, line):
        self.lines.append(line)

    def connectionLost(self, reason):
        if len(self.lines) == 2:
            self.factory.setUser(*self.lines)


class IFingerSetterFactory(Interface):

    def setUser(user, status):
        """
        Return a deferred returning L{bytes}.
        """
```

```
    def buildProtocol(addr):
        """
        Return a protocol returning L{bytes}.
        """


@implementer(IFingerSetterFactory)
class FingerSetterFactoryFromService(protocol.ServerFactory):

    protocol = FingerSetterProtocol

    def __init__(self, service):
        self.service = service

    def setUser(self, user, status):
        self.service.setUser(user, status)


components.registerAdapter(FingerSetterFactoryFromService,
                           IFingerSetterService,
                           IFingerSetterFactory)


class IRCReplyBot(irc.IRCClient):

    def connectionMade(self):
        self.nickname = self.factory.nickname
        irc.IRCClient.connectionMade(self)

    def privmsg(self, user, channel, msg):
        user = user.split('!')[0]
        if self.nickname.lower() == channel.lower():
            d = self.factory.getUser(msg)
            d.addErrback(catchError)
            d.addCallback(lambda m: "Status of %s: %s" % (msg, m))
            d.addCallback(lambda m: self.msg(user, m))


class IIRCClientFactory(Interface):

    """
    @ivar nickname
    """

    def getUser(user):
        """
        Return a deferred returning L{bytes}.
        """

    def buildProtocol(addr):
        """
        Return a protocol.
        """


@implementer(IIRCClientFactory)
class IRCClientFactoryFromService(protocol.ClientFactory):
```

```
    protocol = IRCReplyBot
    nickname = None

    def __init__(self, service):
        self.service = service

    def getUser(self, user):
        return self.service.getUser(user)

components.registerAdapter(IRCClientFactoryFromService,
                           IFingerService,
                           IIRCClientFactory)


class UserStatusTree(resource.Resource):

    template = """<html><head><title>Users</title></head><body>
    <h1>Users</h1>
    <ul>
    %(users)s
    </ul>
    </body>
    </html>"""

    def __init__(self, service):
        resource.Resource.__init__(self)
        self.service = service

    def getChild(self, path, request):
        if path == '':
            return self
        elif path == 'RPC2':
            return UserStatusXR(self.service)
        else:
            return UserStatus(path, self.service)

    def render_GET(self, request):
        users = self.service.getUsers()
        def cbUsers(users):
            request.write(self.template % {'users': ''.join([
                    # Name should be quoted properly these uses.
                    '<li><a href="%s">%s</a></li>' % (name, name)
                    for name in users])})
            request.finish()
        users.addCallback(cbUsers)
        def ebUsers(err):
            log.err(err, "UserStatusTree failed")
            request.finish()
        users.addErrback(ebUsers)
        return server.NOT_DONE_YET

components.registerAdapter(UserStatusTree, IFingerService, resource.IResource)


class UserStatus(resource.Resource):

    template='''<html><head><title>%(title)s</title></head>
```

```
    <body><h1>%(name)s</h1><p>%(status)s</p></body></html>'''

    def __init__(self, user, service):
        resource.Resource.__init__(self)
        self.user = user
        self.service = service

    def render_GET(self, request):
        status = self.service.getUser(self.user)
        def cbStatus(status):
            request.write(self.template % {
                'title': self.user,
                'name': self.user,
                'status': status})
            request.finish()
        status.addCallback(cbStatus)
        def ebStatus(err):
            log.err(err, "UserStatus failed")
            request.finish()
        status.addErrback(ebStatus)
        return server.NOT_DONE_YET


class UserStatusXR(xmlrpc.XMLRPC):

    def __init__(self, service):
        xmlrpc.XMLRPC.__init__(self)
        self.service = service

    def xmlrpc_getUser(self, user):
        return self.service.getUser(user)

    def xmlrpc_getUsers(self):
        return self.service.getUsers()


class IPerspectiveFinger(Interface):

    def remote_getUser(username):
        """
        Return a user's status.
        """

    def remote_getUsers():
        """
        Return a user's status.
        """


@implementer(IPerspectiveFinger)
class PerspectiveFingerFromService(pb.Root):

    def __init__(self, service):
        self.service = service

    def remote_getUser(self, username):
        return self.service.getUser(username)
```

```
    def remote_getUsers(self):
        return self.service.getUsers()

components.registerAdapter(PerspectiveFingerFromService,
                           IFingerService,
                           IPerspectiveFinger)


@implementer(IFingerService)
class FingerService(service.Service):

    def __init__(self, filename):
        self.filename = filename

    def _read(self):
        self.users = {}
        with open(self.filename, "rb") as f:
            for line in f:
                user, status = line.split(b':', 1)
                user = user.strip()
                status = status.strip()
                self.users[user] = status
        self.call = reactor.callLater(30, self._read)

    def getUser(self, user):
        return defer.succeed(self.users.get(user, b"No such user"))

    def getUsers(self):
        return defer.succeed(self.users.keys())

    def startService(self):
        self._read()
        service.Service.startService(self)

    def stopService(self):
        service.Service.stopService(self)
        self.call.cancel()



# Easy configuration

def makeService(config):
    # finger on port 79
    s = service.MultiService()
    f = FingerService(config['file'])
    h = strports.service("tcp:1079", IFingerFactory(f))
    h.setServiceParent(s)


    # website on port 8000
    r = resource.IResource(f)
    r.templateDirectory = config['templates']
    site = server.Site(r)
    j = strports.service("tcp:8000", site)
    j.setServiceParent(s)

    # ssl on port 443
```

```
#     if config.get('ssl'):
#         k = strports.service(
#             "ssl:port=443:certKey=cert.pem:privateKey=key.pem", site
#         )
#         k.setServiceParent(s)

    # irc fingerbot
    if 'ircnick' in config:
        i = IIRCClientFactory(f)
        i.nickname = config['ircnick']
        ircserver = config['ircserver']
        b = internet.ClientService(
            endpoints.HostnameEndpoint(reactor, ircserver, 6667), i
        )
        b.setServiceParent(s)

    # Pespective Broker on port 8889
    if 'pbport' in config:
        m = internet.StreamServerEndpointService(
            endpoints.TCP4ServerEndpoint(reactor, int(config['pbport'])),
            pb.PBServerFactory(IPerspectiveFinger(f))
        )
        m.setServiceParent(s)

    return s
```

tap.py

```
# finger/tap.py
from twisted.application import internet, service
from twisted.internet import interfaces
from twisted.python import usage
import finger

class Options(usage.Options):

    optParameters = [
        ['file', 'f', '/etc/users'],
        ['templates', 't', '/usr/share/finger/templates'],
        ['ircnick', 'n', 'fingerbot'],
        ['ircserver', None, 'irc.freenode.net'],
        ['pbport', 'p', 8889],
        ]

    optFlags = [['ssl', 's']]

def makeService(config):
    return finger.makeService(config)
```

And register it all:

finger_tutorial.py

```

from twisted.application.service import ServiceMaker

finger = ServiceMaker(
    'finger', 'finger.tap', 'Run a finger service', 'finger')
```

Note that the second argument to ServiceMaker ,‘‘finger.tap‘‘ , is a reference to a module (`finger/tap.py`), not to a filename.

And now, the following works

```
% sudo twistd -n finger --file=/etc/users --ircnick=fingerbot
```

For more details about this, see the *twistd plugin documentation* .

Introduction

Twisted is a big system. People are often daunted when they approach it. It's hard to know where to start looking.

This guide builds a full-fledged Twisted application from the ground up, using most of the important bits of the framework. There is a lot of code, but don't be afraid.

The application we are looking at is a "finger" service, along the lines of the familiar service traditionally provided by UNIX™ servers. We will extend this service slightly beyond the standard, in order to demonstrate some of Twisted's higher-level features.

Each section of the tutorial dives straight into applications for various Twisted topics. These topics have their own introductory howtos listed in the *core howto index* and in the documentation for other Twisted projects like Twisted Web and Twisted Words. There are at least three ways to use this tutorial: you may find it useful to read through the rest of the topics listed in the *core howto index* before working through the finger tutorial, work through the finger tutorial and then go back and hit the introductory material that is relevant to the Twisted project you're working on, or read the introductory material one piece at a time as it comes up in the finger tutorial.

Contents

This tutorial is split into eleven parts:

1. *The Evolution of Finger: building a simple finger service*
2. *The Evolution of Finger: adding features to the finger service*
3. *The Evolution of Finger: cleaning up the finger code*
4. *The Evolution of Finger: moving to a component based architecture*
5. *The Evolution of Finger: pluggable backends*
6. *The Evolution of Finger: a web frontend*
7. *The Evolution of Finger: Twisted client support using Perspective Broker*
8. *The Evolution of Finger: using a single factory for multiple protocols*
9. *The Evolution of Finger: a Twisted finger client*
10. *The Evolution of Finger: making a finger library*
11. *The Evolution of Finger: configuration of the finger service*

Setting up the TwistedQuotes application

Goal

This document describes how to set up the TwistedQuotes application used in a number of other documents, such as *designing Twisted applications* .

Setting up the TwistedQuotes project directory

In order to run the Twisted Quotes example, you will need to do the following:

1. Make a `TwistedQuotes` directory on your system
2. Place the following files in the `TwistedQuotes` directory:

   - `__init__.py`

     ```
     """
     Twisted Quotes
     """
     ```

     (this file marks it as a package, see this section of the Python tutorial for more on packages)

   - `quoters.py`

     ```
     from random import choice

     from zope.interface import implementer

     from TwistedQuotes import quoteproto



     @implementer(quoteproto.IQuoter)
     class StaticQuoter:
         """
         Return a static quote.
         """
         def __init__(self, quote):
             self.quote = quote


         def getQuote(self):
             return self.quote



     @implementer(quoteproto.IQuoter)
     class FortuneQuoter:
         """
         Load quotes from a fortune-format file.
         """
         def __init__(self, filenames):
             self.filenames = filenames


         def getQuote(self):
             with open(choice(self.filenames)) as quoteFile:
                 quotes = quoteFile.read().split('\n%\n')
             return choice(quotes)
     ```

   - `quoteproto.py`

     ```
     from zope.interface import Interface

     from twisted.internet.protocol import Factory, Protocol

     ```

```
class IQuoter(Interface):
    """
    An object that returns quotes.
    """
    def getQuote():
        """
        Return a quote.
        """



class QOTD(Protocol):
    def connectionMade(self):
        self.transport.write(self.factory.quoter.getQuote()+'\r\n')
        self.transport.loseConnection()



class QOTDFactory(Factory):
    """
    A factory for the Quote of the Day protocol.

    @type quoter: L{IQuoter} provider
    @ivar quoter: An object which provides L{IQuoter} which will be used by
        the L{QOTD} protocol to get quotes to emit.
    """
    protocol = QOTD

    def __init__(self, quoter):
        self.quoter = quoter
```

3. Add the `TwistedQuotes` directory's *parent* to your Python path. For example, if the TwistedQuotes directory's path is `/mystuff/TwistedQuotes` or `c:\mystuff\TwistedQuotes` add `/mystuff` to your Python path. On UNIX this would be `export PYTHONPATH=/mystuff:$PYTHONPATH` , on Microsoft Windows change the `PYTHONPATH` variable through the Systems Properties dialog by adding `;c:\mystuff` at the end.

4. Test your package by trying to import it in the Python interpreter:

```
Python 2.1.3 (#1, Apr 20 2002, 22:45:31)
[GCC 2.95.4 20011002 (Debian prerelease)] on linux2
Type "copyright", "credits" or "license" for more information.
>>> import TwistedQuotes
>>> # No traceback means you're fine.
```

Designing Twisted Applications

Goals

This document describes how a good Twisted application is structured. It should be useful for beginning Twisted developers who want to structure their code in a clean, maintainable way that reflects current best practices.

Readers will want to be familiar with writing *servers* and *clients* using Twisted.

Example of a modular design: TwistedQuotes

`TwistedQuotes` is a very simple plugin which is a great demonstration of Twisted's power. It will export a small kernel of functionality – Quote of the Day – which can be accessed through every interface that Twisted supports: web pages, e-mail, instant messaging, a specific Quote of the Day protocol, and more.

Set up the project directory

See the description of *setting up the TwistedQuotes example* .

A Look at the Heart of the Application

`quoters.py`

```
from random import choice

from zope.interface import implementer

from TwistedQuotes import quoteproto



@implementer(quoteproto.IQuoter)
class StaticQuoter:
    """
    Return a static quote.
    """
    def __init__(self, quote):
        self.quote = quote


    def getQuote(self):
        return self.quote



@implementer(quoteproto.IQuoter)
class FortuneQuoter:
    """
    Load quotes from a fortune-format file.
    """
    def __init__(self, filenames):
        self.filenames = filenames


    def getQuote(self):
        with open(choice(self.filenames)) as quoteFile:
            quotes = quoteFile.read().split('\n%\n')
        return choice(quotes)
```

This code listing shows us what the Twisted Quotes system is all about. The code doesn't have any way of talking to the outside world, but it provides a library which is a clear and uncluttered abstraction: "give me the quote of the day" .

Note that this module does not import any Twisted functionality at all! The reason for doing things this way is integration. If your "business objects" are not stuck to your user interface, you can make a module that can integrate

those objects with different protocols, GUIs, and file formats. Having such classes provides a way to decouple your components from each other, by allowing each to be used independently.

In this manner, Twisted itself has minimal impact on the logic of your program. Although the Twisted "dot products" are highly interoperable, they also follow this approach. You can use them independently because they are not stuck to each other. They communicate in well-defined ways, and only when that communication provides some additional feature. Thus, you can use twisted.web with twisted.enterprise , but neither requires the other, because they are integrated around the concept of *Deferreds* .

Your Twisted applications should follow this style as much as possible. Have (at least) one module which implements your specific functionality, independent of any user-interface code.

Next, we're going to need to associate this abstract logic with some way of displaying it to the user. We'll do this by writing a Twisted server protocol, which will respond to the clients that connect to it by sending a quote to the client and then closing the connection. Note: don't get too focused on the details of this – different ways to interface with the user are 90% of what Twisted does, and there are lots of documents describing the different ways to do it.

`quoteproto.py`

```
from zope.interface import Interface

from twisted.internet.protocol import Factory, Protocol



class IQuoter(Interface):
    """
    An object that returns quotes.
    """
    def getQuote():
        """
        Return a quote.
        """



class QOTD(Protocol):
    def connectionMade(self):
        self.transport.write(self.factory.quoter.getQuote()+'\r\n')
        self.transport.loseConnection()



class QOTDFactory(Factory):
    """
    A factory for the Quote of the Day protocol.

    @type quoter: L{IQuoter} provider
    @ivar quoter: An object which provides L{IQuoter} which will be used by
        the L{QOTD} protocol to get quotes to emit.
    """
    protocol = QOTD

    def __init__(self, quoter):
        self.quoter = quoter
```

This is a very straightforward `Protocol` implementation, and the pattern described above is repeated here. The Protocol contains essentially no logic of its own, just enough to tie together an object which can generate quotes (a `Quoter`) and an object which can relay bytes to a TCP connection (a `Transport`). When a client connects to this

server, a `QOTD` instance is created, and its `connectionMade` method is called.

The `QOTDFactory` 's role is to specify to the Twisted framework how to create a `Protocol` instance that will handle the connection. Twisted will not instantiate a `QOTDFactory` ; you will do that yourself later, in a `twistd` plug-in.

Note: you can read more specifics of `Protocol` and `Factory` in the *Writing Servers* HOWTO.

Once we have an abstraction – a `Quoter` – and we have a mechanism to connect it to the network – the `QOTD` protocol – the next thing to do is to put the last link in the chain of functionality between abstraction and user. This last link will allow a user to choose a `Quoter` and configure the protocol. Writing this configuration is covered in the *Application HOWTO* .

Overview of Twisted Internet

Twisted Internet is a collection of compatible event-loops for Python. It contains the code to dispatch events to interested observers and a portable API so that observers need not care about which event loop is running. Thus, it is possible to use the same code for different loops, from Twisted's basic, yet portable, `select` -based loop to the loops of various GUI toolkits like GTK+ or Tk.

Twisted Internet contains the various interfaces to the reactor API, whose usage is documented in the low-level chapter. Those APIs are IReactorCore , IReactorTCP , IReactorSSL , IReactorUNIX , IReactorUDP , IReactorTime , IReactorProcess , IReactorMulticast and IReactorThreads . The reactor APIs allow non-persistent calls to be made.

Twisted Internet also covers the interfaces for the various transports, in ITransport and friends. These interfaces allow Twisted network code to be written without regard to the underlying implementation of the transport.

The IProtocolFactory dictates how factories, which are usually a large part of third party code, are written.

Reactor Overview

This HOWTO introduces the Twisted reactor, describes the basics of the reactor and links to the various reactor interfaces.

Reactor Basics

The reactor is the core of the event loop within Twisted – the loop which drives applications using Twisted. The event loop is a programming construct that waits for and dispatches events or messages in a program. It works by calling some internal or external "event provider", which generally blocks until an event has arrived, and then calls the relevant event handler ("dispatches the event"). The reactor provides basic interfaces to a number of services, including network communications, threading, and event dispatching.

For information about using the reactor and the Twisted event loop, see:

- the event dispatching howtos: *Scheduling* and *Using Deferreds* ;
- the communication howtos: *TCP servers* , *TCP clients* , *UDP networking* and *Using processes* ; and
- *Using threads* .

There are multiple implementations of the reactor, each modified to provide better support for specialized features over the default implementation. More information about these and how to use a particular implementation is available via *Choosing a Reactor* .

Twisted applications can use the interfaces in twisted.application.service to configure and run the application instead of using boilerplate reactor code. See *Using Application* for an introduction to Application.

Using the reactor object

You can get to the reactor object using the following code:

```
from twisted.internet import reactor
```

The reactor usually implements a set of interfaces, but depending on the chosen reactor and the platform, some of the interfaces may not be implemented:

- IReactorCore : Core (required) functionality.
- IReactorFDSet : Use FileDescriptor objects.
- IReactorProcess : Process management. Read the *Using Processes* document for more information.
- IReactorSSL : SSL networking support.
- IReactorTCP : TCP networking support. More information can be found in the *Writing Servers* and *Writing Clients* documents.
- IReactorThreads : Threading use and management. More information can be found within *Threading In Twisted* .
- IReactorTime : Scheduling interface. More information can be found within *Scheduling Tasks* .
- IReactorUDP : UDP networking support. More information can be found within *UDP Networking* .
- IReactorUNIX : UNIX socket support.
- IReactorSocket : Third-party socket support.

Using TLS in Twisted

Overview

This document describes how to secure your communications using TLS (Transport Layer Security) — also known as SSL (Secure Sockets Layer) — in Twisted servers and clients. It assumes that you know what TLS is, what some of the major reasons to use it are, and how to generate your own certificates. It also assumes that you are comfortable with creating TCP servers and clients as described in the *server howto* and *client howto* . After reading this document you should be able to create servers and clients that can use TLS to encrypt their connections, switch from using an unencrypted channel to an encrypted one mid-connection, and require client authentication.

Using TLS in Twisted requires that you have pyOpenSSL installed. A quick test to verify that you do is to run `from OpenSSL import SSL` at a python prompt and not get an error.

Twisted provides TLS support as a transport — that is, as an alternative to TCP. When using TLS, use of the TCP APIs you're already familiar with, `TCP4ClientEndpoint` and `TCP4ServerEndpoint` — or `reactor.listenTCP` and `reactor.connectTCP` — is replaced by use of parallel TLS APIs (many of which still use the legacy name "SSL" due to age and/or compatibility with older APIs). To create a TLS server, use SSL4ServerEndpoint or listenSSL . To create a TLS client, use SSL4ClientEndpoint or connectSSL .

TLS provides transport layer security, but it's important to understand what "security" means. With respect to TLS it means three things:

1. Identity: TLS servers (and sometimes clients) present a certificate, offering proof of who they are, so that you know who you are talking to.
2. Confidentiality: once you know who you are talking to, encryption of the connection ensures that the communications can't be understood by any third parties who might be listening in.

3. Integrity: TLS checks the encrypted messages to ensure that they actually came from the party you originally authenticated to. If the messages fail these checks, then they are discarded and your application does not see them.

Without identity, neither confidentiality nor integrity is possible. If you don't know who you're talking to, then you might as easily be talking to your bank or to a thief who wants to steal your bank password. Each of the APIs listed above with "SSL" in the name requires a configuration object called (for historical reasons) a `contextFactory`. (Please pardon the somewhat awkward name.) The `contextFactory` serves three purposes:

1. It provides the materials to prove your own identity to the other side of the connection: in other words, who you are.
2. It expresses your requirements of the other side's identity: in other words, who you would like to talk to (and who you trust to tell you that you're talking to the right party).
3. It allows you to specify certain specialized options about the way the TLS protocol itself operates.

The requirements of clients and servers are slightly different. Both *can* provide a certificate to prove their identity, but commonly, TLS *servers* provide a certificate, whereas TLS *clients* check the server's certificate (to make sure they're talking to the right server) and then later identify themselves to the server some other way, often by offering a shared secret such as a password or API key via an application protocol secured with TLS and not as part of TLS itself.

Since these requirements are slightly different, there are different APIs to construct an appropriate `contextFactory` value for a client or a server.

For servers, we can use twisted.internet.ssl.CertificateOptions. In order to prove the server's identity, you pass the `privateKey` and `certificate` arguments to this object. twisted.internet.ssl.PrivateCertificate.options is a convenient way to create a `CertificateOptions` instance configured to use a particular key and certificate.

For clients, we can use twisted.internet.ssl.optionsForClientTLS. This takes two arguments, `hostname` (which indicates what hostname must be advertised in the server's certificate) and optionally `trustRoot`. By default, optionsForClientTLS tries to obtain the trust roots from your platform, but you can specify your own.

You may obtain an object suitable to pass as the `trustRoot=` parameter with an explicit list of twisted.internet.ssl.Certificate or twisted.internet.ssl.PrivateCertificate instances by calling twisted.internet.ssl.trustRootFromCertificates. This will cause optionsForClientTLS to accept any connection so long as the server's certificate is signed by at least one of the certificates passed.

Note: Currently, Twisted only supports loading of OpenSSL's default trust roots. If you've built OpenSSL yourself, you must take care to include these in the appropriate location. If you're using the OpenSSL shipped as part of Mac OS X 10.5-10.9, this behavior will also be correct. If you're using Debian, or one of its derivatives like Ubuntu, install the *ca-certificates* package to ensure you have trust roots available, and this behavior should also be correct. Work is ongoing to make platformTrust — the API that optionsForClientTLS uses by default — more robust. For example, platformTrust should fall back to the "certifi" package if no platform trust roots are available but it doesn't do that yet. When this happens, you shouldn't need to change your code.

TLS echo server and client

Now that we've got the theory out of the way, let's try some working examples of how to get started with a TLS server. The following examples rely on the files `server.pem` (private key and self-signed certificate together) and `public.pem` (the server's public certificate by itself).

TLS echo server

`echoserv_ssl.py`

```
#!/usr/bin/env python
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

import sys

from twisted.internet import ssl, protocol, task, defer
from twisted.python import log
from twisted.python.modules import getModule

import echoserv

def main(reactor):
    log.startLogging(sys.stdout)
    certData = getModule(__name__).filePath.sibling('server.pem').getContent()
    certificate = ssl.PrivateCertificate.loadPEM(certData)
    factory = protocol.Factory.forProtocol(echoserv.Echo)
    reactor.listenSSL(8000, factory, certificate.options())
    return defer.Deferred()

if __name__ == '__main__':
    import echoserv_ssl
    task.react(echoserv_ssl.main)
```

This server uses listenSSL to listen for TLS traffic on port 8000, using the certificate and private key contained in the file `server.pem`. It uses the same echo example server as the TCP echo server — even going so far as to import its protocol class. Assuming that you can buy your own TLS certificate from a certificate authority, this is a fairly realistic TLS server.

TLS echo client

`echoclient_ssl.py`

```
#!/usr/bin/env python
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from twisted.internet import ssl, task, protocol, endpoints, defer
from twisted.python.modules import getModule

import echoclient

@defer.inlineCallbacks
def main(reactor):
    factory = protocol.Factory.forProtocol(echoclient.EchoClient)
    certData = getModule(__name__).filePath.sibling('public.pem').getContent()
    authority = ssl.Certificate.loadPEM(certData)
    options = ssl.optionsForClientTLS(u'example.com', authority)
    endpoint = endpoints.SSL4ClientEndpoint(reactor, 'localhost', 8000,
                                            options)
    echoClient = yield endpoint.connect(factory)

    done = defer.Deferred()
    echoClient.connectionLost = lambda reason: done.callback(None)
    yield done
```

```
if __name__ == '__main__':
    import echoclient_ssl
    task.react(echoclient_ssl.main)
```

This client uses SSL4ClientEndpoint to connect to `echoserv_ssl.py`. It *also* uses the same echo example client as the TCP echo client. Whenever you have a protocol that listens on plain-text TCP it is easy to run it over TLS instead. It specifies that it only wants to talk to a host named `"example.com"`, and that it trusts the certificate authority in `"public.pem"` to say who `"example.com"` is. Note that the host you are connecting to — localhost — and the host whose identity you are verifying — example.com — can differ. In this case, our example `server.pem` certificate identifies a host named "example.com", but your server is proably running on localhost.

In a realistic client, it's very important that you pass the same "hostname" your connection API (in this case, SSL4ClientEndpoint) and optionsForClientTLS. In this case we're using "`localhost`" as the host to connect to because you're probably running this example on your own computer and "`example.com`" because that's the value hard-coded in the dummy certificate distributed along with Twisted's example code.

Connecting To Public Servers

Here is a short example, now using the default trust roots for optionsForClientTLS from platformTrust.

`check_server_certificate.py`

```
from __future__ import print_function
import sys
from twisted.internet import defer, endpoints, protocol, ssl, task, error

def main(reactor, host, port=443):
    options = ssl.optionsForClientTLS(hostname=host.decode('utf-8'))
    port = int(port)

    class ShowCertificate(protocol.Protocol):
        def connectionMade(self):
            self.transport.write(b"GET / HTTP/1.0\r\n\r\n")
            self.done = defer.Deferred()
        def dataReceived(self, data):
            certificate = ssl.Certificate(self.transport.getPeerCertificate())
            print("OK:", certificate)
            self.transport.abortConnection()
        def connectionLost(self, reason):
            print("Lost.")
            if not reason.check(error.ConnectionClosed):
                print("BAD:", reason.value)
            self.done.callback(None)

    return endpoints.connectProtocol(
        endpoints.SSL4ClientEndpoint(reactor, host, port, options),
        ShowCertificate()
    ).addCallback(lambda protocol: protocol.done)

task.react(main, sys.argv[1:])
```

You can use this tool fairly simply to retrieve certificates from an HTTPS server with a valid TLS certificate, by running it with a host name. For example:

```
$ python check_server_certificate.py www.twistedmatrix.com
OK: <Certificate Subject=www.twistedmatrix.com ...>
```

```
$ python check_server_certificate.py www.cacert.org
BAD: [(... 'certificate verify failed')]
$ python check_server_certificate.py dornkirk.twistedmatrix.com
BAD: No service reference ID could be validated against certificate.
```

Note: To *properly* validate your `hostname` parameter according to RFC6125, please also install the "service_identity" and "idna" packages from PyPI. Without this package, Twisted will currently make a conservative guess as to the correctness of the server's certificate, but this will reject a large number of potentially valid certificates. *service_identity* implements the standard correctly and it will be a required dependency for TLS in a future release of Twisted.

Using startTLS

If you want to switch from unencrypted to encrypted traffic mid-connection, you'll need to turn on TLS with startTLS on both ends of the connection at the same time via some agreed-upon signal like the reception of a particular message. You can readily verify the switch to an encrypted channel by examining the packet payloads with a tool like Wireshark .

startTLS server

`starttls_server.py`

```
from __future__ import print_function

from twisted.internet import ssl, protocol, defer, task, endpoints
from twisted.protocols.basic import LineReceiver
from twisted.python.modules import getModule

class TLSServer(LineReceiver):
    def lineReceived(self, line):
        print("received: ", line)
        if line == b"STARTTLS":
            print("-- Switching to TLS")
            self.sendLine(b'READY')
            self.transport.startTLS(self.factory.options)

def main(reactor):
    certData = getModule(__name__).filePath.sibling('server.pem').getContent()
    cert = ssl.PrivateCertificate.loadPEM(certData)
    factory = protocol.Factory.forProtocol(TLSServer)
    factory.options = cert.options()
    endpoint = endpoints.TCP4ServerEndpoint(reactor, 8000)
    endpoint.listen(factory)
    return defer.Deferred()

if __name__ == '__main__':
    import starttls_server
    task.react(starttls_server.main)
```

startTLS client

starttls_client.py

```
from __future__ import print_function

from twisted.internet import ssl, endpoints, task, protocol, defer
from twisted.protocols.basic import LineReceiver
from twisted.python.modules import getModule

class StartTLSClient(LineReceiver):
    def connectionMade(self):
        self.sendLine(b"plain text")
        self.sendLine(b"STARTTLS")

    def lineReceived(self, line):
        print("received: ", line)
        if line == b"READY":
            self.transport.startTLS(self.factory.options)
            self.sendLine(b"secure text")
            self.transport.loseConnection()

@defer.inlineCallbacks
def main(reactor):
    factory = protocol.Factory.forProtocol(StartTLSClient)
    certData = getModule(__name__).filePath.sibling('server.pem').getContent()
    factory.options = ssl.optionsForClientTLS(
        u"example.com", ssl.PrivateCertificate.loadPEM(certData)
    )
    endpoint = endpoints.HostnameEndpoint(reactor, 'localhost', 8000)
    startTLSClient = yield endpoint.connect(factory)

    done = defer.Deferred()
    startTLSClient.connectionLost = lambda reason: done.callback(None)
    yield done

if __name__ == "__main__":
    import starttls_client
    task.react(starttls_client.main)
```

`startTLS` is a transport method that gets passed a `contextFactory`. It is invoked at an agreed-upon time in the data reception method of the client and server protocols. The server uses `PrivateCertificate.options` to create a `contextFactory` which will use a particular certificate and private key (a common requirement for TLS servers).

The client creates an uncustomized `CertificateOptions` which is all that's necessary for a TLS client to interact with a TLS server.

Client authentication

Server and client-side changes to require client authentication fall largely under the dominion of pyOpenSSL, but few examples seem to exist on the web so for completeness a sample server and client are provided here.

TLS server with client authentication via client certificate verification

When one or more certificates are passed to `PrivateCertificate.options`, the resulting `contextFactory` will use those certificates as trusted authorities and require that the peer present a certificate with a valid chain anchored by one of those authorities.

A server can use this to verify that a client provides a valid certificate signed by one of those certificate authorities; here is an example of such a certificate.

`ssl_clientauth_server.py`

```
#!/usr/bin/env python
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

import sys

from twisted.internet import ssl, protocol, task, defer
from twisted.python import log
from twisted.python.modules import getModule

import echoserv

def main(reactor):
    log.startLogging(sys.stdout)
    certData = getModule(__name__).filePath.sibling('public.pem').getContent()
    authData = getModule(__name__).filePath.sibling('server.pem').getContent()
    authority = ssl.Certificate.loadPEM(certData)
    certificate = ssl.PrivateCertificate.loadPEM(authData)
    factory = protocol.Factory.forProtocol(echoserv.Echo)
    reactor.listenSSL(8000, factory, certificate.options(authority))
    return defer.Deferred()

if __name__ == '__main__':
    import ssl_clientauth_server
    task.react(ssl_clientauth_server.main)
```

Client with certificates

The following client then supplies such a certificate as the `clientCertificate` argument to optionsForClientTLS, while still validating the server's identity.

`ssl_clientauth_client.py`

```
#!/usr/bin/env python
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from twisted.internet import ssl, task, protocol, endpoints, defer
from twisted.python.modules import getModule

import echoclient

@defer.inlineCallbacks
def main(reactor):
    factory = protocol.Factory.forProtocol(echoclient.EchoClient)
    certData = getModule(__name__).filePath.sibling('public.pem').getContent()
```

```
    authData = getModule(__name__).filePath.sibling('server.pem').getContent()
    clientCertificate = ssl.PrivateCertificate.loadPEM(authData)
    authority = ssl.Certificate.loadPEM(certData)
    options = ssl.optionsForClientTLS(u'example.com', authority,
                                      clientCertificate)
    endpoint = endpoints.SSL4ClientEndpoint(reactor, 'localhost', 8000,
                                            options)
    echoClient = yield endpoint.connect(factory)

    done = defer.Deferred()
    echoClient.connectionLost = lambda reason: done.callback(None)
    yield done

if __name__ == '__main__':
    import ssl_clientauth_client
    task.react(ssl_clientauth_client.main)
```

Notice that these two examples are very, very similar to the TLS echo examples above. In fact, you can demonstrate a failed authentication by simply running `echoclient_ssl.py` against `ssl_clientauth_server.py`; you'll see no output because the server closed the connection rather than echoing the client's authenticated input.

TLS Protocol Options

For servers, it is desirable to offer Diffie-Hellman based key exchange that provides perfect forward secrecy. The ciphers are activated by default, however it is necessary to pass an instance of DiffieHellmanParameters to `CertificateOptions` via the `dhParameters` option to be able to use them.

For example,

```
from twisted.internet.ssl import CertificateOptions, DiffieHellmanParameters
from twisted.python.filepath import FilePath
dhFilePath = FilePath('dh_param_1024.pem')
dhParams = DiffieHellmanParameters.fromFile(dhFilePath)
options = CertificateOptions(..., dhParameters=dhParams)
```

Another part of the TLS protocol which `CertificateOptions` can control is the version of the TLS or SSL protocol used. By default, Twisted will configure it to use TLSv1.0 or later and disable the insecure SSLv3 protocol. Manual control over protocols can be helpful if you need to support legacy SSLv3 systems, or you wish to restrict it down to just the strongest of the TLS versions.

You can ask `CertificateOptions` to use a more secure default minimum than Twisted's by using the `raiseMinimumTo` argument in the initializer:

```
from twisted.internet.ssl import CertificateOptions, TLSVersion
options = CertificateOptions(
    ...,
    raiseMinimumTo=TLSVersion.TLSv1_1)
```

This will always negotiate a minimum of TLSv1.1, but will negotiate higher versions if Twisted's default is higher. This usage will stay secure if Twisted updates the minimum to TLSv1.2, rather than causing your application to use the now theoretically insecure minimum you set.

If you need a strictly hard range of TLS versions you wish `CertificateOptions` to negotiate, you can use the `insecurelyLowerMinimumTo` and `lowerMaximumSecurityTo` arguments in the initializer:

```
from twisted.internet.ssl import CertificateOptions, TLSVersion
options = CertificateOptions(
```

```
    ...,
    insecurelyLowerMinimumTo=TLSVersion.TLSv1_0,
    lowerMaximumSecurityTo=TLSVersion.TLSv1_2)
```

This will cause it to negotiate between TLSv1.0 and TLSv1.2, and will not change if Twisted's default minimum TLS version is raised. It is highly recommended not to set `lowerMaximumSecurityTo` unless you have a peer that is known to misbehave on newer TLS versions, and to only set `insecurelyLowerMinimumTo` when Twisted's minimum is not acceptable. Using these two arguments to `CertificateOptions` may make your application's TLS insecure if you do not review it frequently, and should not be used in libraries.

SSLv3 support is still available and you can enable support for it if you wish. As an example, this supports all TLS versions and SSLv3:

```
from twisted.internet.ssl import CertificateOptions, TLSVersion
options = CertificateOptions(
    ...,
    insecurelyLowerMinimumTo=TLSVersion.SSLv3)
```

Future OpenSSL versions may completely remove the ability to negotiate the insecure SSLv3 protocol, and this will not allow you to re-enable it.

Additionally, it is possible to limit the acceptable ciphers for your connection by passing an IAcceptableCiphers object to `CertificateOptions`. Since Twisted uses a secure cipher configuration by default, it is discouraged to do so unless absolutely necessary.

Application Layer Protocol Negotiation (ALPN) and Next Protocol Negotiation (NPN)

ALPN and NPN are TLS extensions that can be used by clients and servers to negotiate what application-layer protocol will be spoken once the encrypted connection is established. This avoids the need for extra custom round trips once the encrypted connection is established. It is implemented as a standard part of the TLS handshake.

NPN is supported from OpenSSL version 1.0.1. ALPN is the newer of the two protocols, supported in OpenSSL versions 1.0.2 onward. These functions require pyOpenSSL version 0.15 or higher. To query the methods supported by your system, use twisted.internet.ssl.protocolNegotiationMechanisms. It will return a collection of flags indicating support for NPN and/or ALPN.

twisted.internet.ssl.CertificateOptions and twisted.internet.ssl.optionsForClientTLS allow for selecting the protocols your program is willing to speak after the connection is established.

On the server=side you will have:

```
from twisted.internet.ssl import CertificateOptions
options = CertificateOptions(..., acceptableProtocols=[b'h2', b'http/1.1'])
```

and for clients:

```
from twisted.internet.ssl import optionsForClientTLS
options = optionsForClientTLS(hostname=hostname, acceptableProtocols=[b'h2', b'http/1.
↪1'])
```

Twisted will attempt to use both ALPN and NPN, if they're available, to maximise compatibility with peers. If both ALPN and NPN are supported by the peer, the result from ALPN is preferred.

For NPN, the client selects the protocol to use; For ALPN, the server does. If Twisted is acting as the peer who is supposed to select the protocol, it will prefer the earliest protocol in the list that is supported by both peers.

To determine what protocol was negotiated, after the connection is done, use TLSMemoryBIOProtocol.negotiatedProtocol. It will return one of the protocol names passed to the `acceptableProtocols` parameter. It will return `None` if the peer did not offer ALPN or NPN.

It can also return `None` if no overlap could be found and the connection was established regardless (some peers will do this: Twisted will not). In this case, the protocol that should be used is whatever protocol would have been used if negotiation had not been attempted at all.

> **Warning:** If ALPN or NPN are used and no overlap can be found, then the remote peer may choose to terminate the connection. This may cause the TLS handshake to fail, or may result in the connection being torn down immediately after being made. If Twisted is the selecting peer (that is, Twisted is the server and ALPN is being used, or Twisted is the client and NPN is being used), and no overlap can be found, Twisted will always choose to fail the handshake rather than allow an ambiguous connection to set up.

An example of using this functionality can be found in `this example script for clients` and `this example script for servers`.

Related facilities

twisted.protocols.amp supports encrypted connections and exposes a `startTLS` method one can use or subclass. twisted.web has built-in TLS support in its client , http , and xmlrpc modules.

Conclusion

After reading through this tutorial, you should be able to:

- Use `listenSSL` and `connectSSL` to create servers and clients that use TLS
- Use `startTLS` to switch a channel from being unencrypted to using TLS mid-connection
- Add server and client support for client authentication

UDP Networking

Overview

Unlike TCP, UDP has no notion of connections. A UDP socket can receive datagrams from any server on the network and send datagrams to any host on the network. In addition, datagrams may arrive in any order, never arrive at all, or be duplicated in transit.

Since there are no connections, we only use a single object, a protocol, for each UDP socket. We then use the reactor to connect this protocol to a UDP transport, using the twisted.internet.interfaces.IReactorUDP reactor API.

DatagramProtocol

The class where you actually implement the protocol parsing and handling will usually be descended from twisted.internet.protocol.DatagramProtocol or from one of its convenience children. The `DatagramProtocol` class receives datagrams and can send them out over the network. Received datagrams include the address they were sent from. When sending datagrams the destination address must be specified.

Here is a simple example:

`basic_example.py`

```
from __future__ import print_function

from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor


class Echo(DatagramProtocol):

    def datagramReceived(self, data, addr):
        print("received %r from %s" % (data, addr))
        self.transport.write(data, addr)

reactor.listenUDP(9999, Echo())
reactor.run()
```

As you can see, the protocol is registered with the reactor. This means it may be persisted if it's added to an application, and thus it has startProtocol and stopProtocol methods that will get called when the protocol is connected and disconnected from a UDP socket.

The protocol's `transport` attribute will implement the twisted.internet.interfaces.IUDPTransport interface. Notice that `addr` argument to `self.transport.write` should be a tuple with IP address and port number. First element of tuple must be ip address and not a hostname. If you only have the hostname use `reactor.resolve()` to resolve the address (see twisted.internet.interfaces.IReactorCore.resolve).

Other thing to keep in mind is that data written to transport must be bytes. Trying to write string may work ok in Python 2, but will fail if you are using Python 3.

To confirm that socket is indeed listening you can try following command line one-liner.

```
> echo "Hello World!" | nc -4u -w1 localhost 9999
```

If everything is ok your "server" logs should print:

```
received b'Hello World!\n' from ('127.0.0.1', 32844) # where 32844 is some random␣
↪port number
```

Adopting Datagram Ports

By default `reactor.listenUDP()` call will create appropriate socket for you, but it is also possible to add an existing `SOCK_DGRAM` file descriptor of some socket to the reactor using the adoptDatagramPort API.

Here is a simple example:

`adopt_datagram_port.py`

```
from __future__ import print_function

import socket

from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor


class Echo(DatagramProtocol):
    def datagramReceived(self, data, addr):
        print("received %r from %s" % (data, addr))
        self.transport.write(data, addr)
```

```
# Create new socket that will be passed to reactor later.
portSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Make the port non-blocking and start it listening.
portSocket.setblocking(False)
portSocket.bind(('127.0.0.1', 9999))

# Now pass the port file descriptor to the reactor.
port = reactor.adoptDatagramPort(
    portSocket.fileno(), socket.AF_INET, Echo())

# The portSocket should be cleaned up by the process that creates it.
portSocket.close()

reactor.run()
```

Note:

- You must ensure that the socket is non-blocking before passing its file descriptor to adoptDatagramPort.
- adoptDatagramPort cannot (currently) detect the family of the adopted socket so you must ensure that you pass the correct socket family argument.
- The reactor will not shutdown the socket. It is the responsibility of the process that created the socket to shutdown and clean up the socket when it is no longer needed.

Connected UDP

A connected UDP socket is slightly different from a standard one as it can only send and receive datagrams to/from a single address. However this does not in any way imply a connection as datagrams may still arrive in any order and the port on the other side may have no one listening. The benefit of the connected UDP socket is that it **may** provide notification of undelivered packages. This depends on many factors (almost all of which are out of the control of the application) but still presents certain benefits which occasionally make it useful.

Unlike a regular UDP protocol, we do not need to specify where to send datagrams and are not told where they came from since they can only come from the address to which the socket is 'connected'.

`connected_udp.py`

```
from __future__ import print_function

from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor


class Helloer(DatagramProtocol):
    def startProtocol(self):
        host = "192.168.1.1"
        port = 1234

        self.transport.connect(host, port)
        print(("now we can only send to host %s port %d" % (host, port)))
        self.transport.write(b"hello")  # no need for address

    def datagramReceived(self, data, addr):
```

```
        print("received %r from %s" % (data, addr))

    # Possibly invoked if there is no server listening on the
    # address to which we are sending.
    def connectionRefused(self):
        print("No one listening")

# 0 means any port, we don't care in this case
reactor.listenUDP(0, Helloer())
reactor.run()
```

Note that `connect()`, like `write()` will only accept IP addresses, not unresolved hostnames. To obtain the IP of a hostname use `reactor.resolve()`, e.g:

`getting_ip.py`

```
from __future__ import print_function

from twisted.internet import reactor


def gotIP(ip):
    print("IP of 'localhost' is", ip)
    reactor.stop()

reactor.resolve('localhost').addCallback(gotIP)
reactor.run()
```

Connecting to a new address after a previous connection or making a connected port unconnected are not currently supported, but likely will be in the future.

Multicast UDP

Multicast allows a process to contact multiple hosts with a single packet, without knowing the specific IP address of any of the hosts. This is in contrast to normal, or unicast, UDP, where each datagram has a single IP as its destination. Multicast datagrams are sent to special multicast group addresses (in the IPv4 range 224.0.0.0 to 239.255.255.255), along with a corresponding port. In order to receive multicast datagrams, you must join that specific group address. However, any UDP socket can send to multicast addresses. Here is a simple server example:

`MulticastServer.py`

```
from __future__ import print_function

from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor


class MulticastPingPong(DatagramProtocol):

    def startProtocol(self):
        """
        Called after protocol has started listening.
        """
        # Set the TTL>1 so multicast will cross router hops:
        self.transport.setTTL(5)
        # Join a specific multicast group:
        self.transport.joinGroup("228.0.0.5")
```

```
    def datagramReceived(self, datagram, address):
        print("Datagram %s received from %s" % (repr(datagram), repr(address)))
        if datagram == b"Client: Ping" or datagram == "Client: Ping":
            # Rather than replying to the group multicast address, we send the
            # reply directly (unicast) to the originating port:
            self.transport.write(b"Server: Pong", address)


# We use listenMultiple=True so that we can run MulticastServer.py and
# MulticastClient.py on same machine:
reactor.listenMulticast(9999, MulticastPingPong(),
                        listenMultiple=True)
reactor.run()
```

As with UDP, with multicast there is no server/client differentiation at the protocol level. Our server example is very simple and closely resembles a normal listenUDP protocol implementation. The main difference is that instead of `listenUDP`, listenMulticast is called with the port number. The server calls joinGroup to join a multicast group. A `DatagramProtocol` that is listening with multicast and has joined a group can receive multicast datagrams, but also unicast datagrams sent directly to its address. The server in the example above sends such a unicast message in reply to the multicast message it receives from the client.

Client code may look like this:

`MulticastClient.py`

```
from __future__ import print_function

from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor


class MulticastPingClient(DatagramProtocol):

    def startProtocol(self):
        # Join the multicast address, so we can receive replies:
        self.transport.joinGroup("228.0.0.5")
        # Send to 228.0.0.5:9999 - all listeners on the multicast address
        # (including us) will receive this message.
        self.transport.write(b'Client: Ping', ("228.0.0.5", 9999))

    def datagramReceived(self, datagram, address):
        print("Datagram %s received from %s" % (repr(datagram), repr(address)))


reactor.listenMulticast(9999, MulticastPingClient(), listenMultiple=True)
reactor.run()
```

Note that a multicast socket will have a default TTL (time to live) of 1. That is, datagrams won't traverse more than one router hop, unless a higher TTL is set with setTTL. Other functionality provided by the multicast transport includes setOutgoingInterface and setLoopbackMode – see IMulticastTransport for more information.

To test your multicast setup you need to start server in one terminal and couple of clients in other terminals. If all goes ok you should see "Ping" messages sent by each client in logs of all other connected clients.

Broadcast UDP

Broadcast allows a different way of contacting several unknown hosts. Broadcasting via UDP sends a packet out to all hosts on the local network by sending to a magic broadcast address (`"<broadcast>"`). This broadcast is filtered by routers by default, and there are no "groups" like multicast, only different ports.

Broadcast is enabled by passing `True` to setBroadcastAllowed on the port. Checking the broadcast status can be done with getBroadcastAllowed on the port.

For a complete example of this feature, see `udpbroadcast.py`.

IPv6

UDP sockets can also bind to IPv6 addresses to support sending and receiving datagrams over IPv6. By passing an IPv6 address to listenUDP's `interface` argument, the reactor will start an IPv6 socket that can be used to send and receive UDP datagrams.

`ipv6_listen.py`

```
from __future__ import print_function

from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor


class Echo(DatagramProtocol):
    def datagramReceived(self, data, addr):
        print("received %r from %s" % (data, addr))
        self.transport.write(data, addr)

reactor.listenUDP(9999, Echo(), interface='::')
reactor.run()
```

Using Processes

Overview

Along with connection to servers across the internet, Twisted also connects to local processes with much the same API. The API is described in more detail in the documentation of:

- twisted.internet.interfaces.IReactorProcess
- twisted.internet.interfaces.IProcessTransport
- twisted.internet.interfaces.IProcessProtocol

Running Another Process

Processes are run through the reactor, using `reactor.spawnProcess` . Pipes are created to the child process, and added to the reactor core so that the application will not block while sending data into or pulling data out of the new process. `reactor.spawnProcess` requires two arguments, `processProtocol` and `executable` , and optionally takes several more: `args` , `environment` , `path` , `userID` , `groupID` , `usePTY` , and `childFDs` . Not all of these are available on Windows.

```
from twisted.internet import reactor

processProtocol = MyProcessProtocol()
reactor.spawnProcess(processProtocol, executable, args=[program, arg1, arg2],
                     env={'HOME': os.environ['HOME']}, path,
                     uid, gid, usePTY, childFDs)
```

- `processProtocol` should be an instance of a subclass of twisted.internet.protocol.ProcessProtocol . The interface is described below.
- `executable` is the full path of the program to run. It will be connected to processProtocol.
- `args` is a list of command line arguments to be passed to the process. `args[0]` should be the name of the process.
- `env` is a dictionary containing the environment to pass through to the process.
- `path` is the directory to run the process in. The child will switch to the given directory just before starting the new program. The default is to stay in the current directory.
- `uid` and `gid` are the user ID and group ID to run the subprocess as. Of course, changing identities will be more likely to succeed if you start as root.
- `usePTY` specifies whether the child process should be run with a pty, or if it should just get a pair of pipes. Whether a program needs to be run with a PTY or not depends on the particulars of that program. Often, programs which primarily interact with users via a terminal do need a PTY.
- `childFDs` lets you specify how the child's file descriptors should be set up. Each key is a file descriptor number (an integer) as seen by the child. 0, 1, and 2 are usually stdin, stdout, and stderr, but some programs may be instructed to use additional fds through command-line arguments or environment variables. Each value is either an integer specifying one of the parent's current file descriptors, the string "r" which creates a pipe that the parent can read from, or the string "w" which creates a pipe that the parent can write to. If `childFDs` is not provided, a default is used which creates the usual stdin-writer, stdout-reader, and stderr-reader pipes.

`args` and `env` have empty default values, but many programs depend upon them to be set correctly. At the very least, `args[0]` should probably be the same as `executable` . If you just provide `os.environ` for `env` , the child program will inherit the environment from the current process, which is usually the civilized thing to do (unless you want to explicitly clean the environment as a security precaution). The default is to give an empty `env` to the child.

`reactor.spawnProcess` returns an instance that implements IProcessTransport.

Writing a ProcessProtocol

The ProcessProtocol you pass to `spawnProcess` is your interaction with the process. It has a very similar signature to a regular Protocol, but it has several extra methods to deal with events specific to a process. In our example, we will interface with 'wc' to create a word count of user-given text. First, we'll start by importing the required modules, and writing the initialization for our ProcessProtocol.

```
from twisted.internet import protocol
class WCProcessProtocol(protocol.ProcessProtocol):

    def __init__(self, text):
        self.text = text
```

When the ProcessProtocol is connected to the protocol, it has the connectionMade method called. In our protocol, we will write our text to the standard input of our process and then close standard input, to let the process know we are done writing to it.

```
...
    def connectionMade(self):
        self.transport.write(self.text)
        self.transport.closeStdin()
```

At this point, the process has received the data, and it's time for us to read the results. Instead of being received in `dataReceived`, data from standard output is received in `outReceived`. This is to distinguish it from data on standard error.

```
...
    def outReceived(self, data):
        fieldLength = len(data) / 3
        lines = int(data[:fieldLength])
        words = int(data[fieldLength:fieldLength*2])
        chars = int(data[fieldLength*2:])
        self.transport.loseConnection()
        self.receiveCounts(lines, words, chars)
```

Now, the process has parsed the output, and ended the connection to the process. Then it sends the results on to the final method, receiveCounts. This is for users of the class to override, so as to do other things with the data. For our demonstration, we will just print the results.

```
...
    def receiveCounts(self, lines, words, chars):
        print('Received counts from wc.')
        print('Lines:', lines)
        print('Words:', words)
        print('Characters:', chars)
```

We're done! To use our WCProcessProtocol, we create an instance, and pass it to spawnProcess.

```
from twisted.internet import reactor
wcProcess = WCProcessProtocol("accessing protocols through Twisted is fun!\n")
reactor.spawnProcess(wcProcess, 'wc', ['wc'])
reactor.run()
```

Things that can happen to your ProcessProtocol

These are the methods that you can usefully override in your subclass of `ProcessProtocol`:

- `.connectionMade()`: This is called when the program is started, and makes a good place to write data into the stdin pipe (using `self.transport.write`).
- `.outReceived(data)`: This is called with data that was received from the process' stdout pipe. Pipes tend to provide data in larger chunks than sockets (one kilobyte is a common buffer size), so you may not experience the "random dribs and drabs" behavior typical of network sockets, but regardless you should be prepared to deal if you don't get all your data in a single call. To do it properly, `outReceived` ought to simply accumulate the data and put off doing anything with it until the process has finished.
- `.errReceived(data)`: This is called with data from the process' stderr pipe. It behaves just like `outReceived`.
- `.inConnectionLost`: This is called when the reactor notices that the process' stdin pipe has closed. Programs don't typically close their own stdin, so this will probably get called when your ProcessProtocol has shut down the write side with `self.transport.loseConnection`.

- `.outConnectionLost` : This is called when the program closes its stdout pipe. This usually happens when the program terminates.
- `.errConnectionLost` : Same as `outConnectionLost` , but for stderr instead of stdout.
- `.processExited(status)` : This is called when the child process has been reaped, and receives information about the process' exit status. The status is passed in the form of a Failure instance, created with a `.value` that either holds a ProcessDone object if the process terminated normally (it died of natural causes instead of receiving a signal, and if the exit code was 0), or a ProcessTerminated object (with an `.exitCode` attribute) if something went wrong.
- `.processEnded(status)` : This is called when all the file descriptors associated with the child process have been closed and the process has been reaped. This means it is the last callback which will be made onto a `ProcessProtocol` . The `status` parameter has the same meaning as it does for `processExited` .

The base-class definitions of most of these functions are no-ops. This will result in all stdout and stderr being thrown away. Note that it is important for data you don't care about to be thrown away: if the pipe were not read, the child process would eventually block as it tried to write to a full pipe.

Things you can do from your ProcessProtocol

The following are the basic ways to control the child process:

- `self.transport.write(data)` : Stuff some data in the stdin pipe. Note that this `write` method will queue any data that can't be written immediately. Writing will resume in the future when the pipe becomes writable again.
- `self.transport.closeStdin` : Close the stdin pipe. Programs which act as filters (reading from stdin, modifying the data, writing to stdout) usually take this as a sign that they should finish their job and terminate. For these programs, it is important to close stdin when you're done with it, otherwise the child process will never quit.
- `self.transport.closeStdout` : Not usually called, since you're putting the process into a state where any attempt to write to stdout will cause a SIGPIPE error. This isn't a nice thing to do to the poor process.
- `self.transport.closeStderr` : Not usually called, same reason as `closeStdout` .
- `self.transport.loseConnection` : Close all three pipes.
- `self.transport.signalProcess('KILL')` : Kill the child process. This will eventually result in `processEnded` being called.

Verbose Example

Here is an example that is rather verbose about exactly when all the methods are called. It writes a number of lines into the `wc` program and then parses the output.

`process.py`

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from twisted.internet import protocol
from twisted.internet import reactor
import re
```

```
class MyPP(protocol.ProcessProtocol):
    def __init__(self, verses):
        self.verses = verses
        self.data = ""
    def connectionMade(self):
        print("connectionMade!")
        for i in range(self.verses):
            self.transport.write("Aleph-null bottles of beer on the wall,\n" +
                                 "Aleph-null bottles of beer,\n" +
                                 "Take one down and pass it around,\n" +
                                 "Aleph-null bottles of beer on the wall.\n")
        self.transport.closeStdin() # tell them we're done
    def outReceived(self, data):
        print("outReceived! with %d bytes!" % len(data))
        self.data = self.data + data
    def errReceived(self, data):
        print("errReceived! with %d bytes!" % len(data))
    def inConnectionLost(self):
        print("inConnectionLost! stdin is closed! (we probably did it)")
    def outConnectionLost(self):
        print("outConnectionLost! The child closed their stdout!")
        # now is the time to examine what they wrote
        #print("I saw them write:", self.data)
        (dummy, lines, words, chars, file) = re.split(r'\s+', self.data)
        print("I saw %s lines" % lines)
    def errConnectionLost(self):
        print("errConnectionLost! The child closed their stderr.")
    def processExited(self, reason):
        print("processExited, status %d" % (reason.value.exitCode,))
    def processEnded(self, reason):
        print("processEnded, status %d" % (reason.value.exitCode,))
        print("quitting")
        reactor.stop()

pp = MyPP(10)
reactor.spawnProcess(pp, "wc", ["wc"], {})
reactor.run()
```

The exact output of this program depends upon the relative timing of some un-synchronized events. In particular, the program may observe the child process close its stderr pipe before or after it reads data from the stdout pipe. One possible transcript would look like this:

```
% ./process.py
connectionMade!
inConnectionLost! stdin is closed! (we probably did it)
errConnectionLost! The child closed their stderr.
outReceived! with 24 bytes!
outConnectionLost! The child closed their stdout!
I saw 40 lines
processEnded, status 0
quitting
Main loop terminated.
%
```

Doing it the Easy Way

Frequently, one just needs a simple way to get all the output from a program. In the blocking world, you might use `commands.getoutput` from the standard library, but using that in an event-driven program will cause everything else to stall until the command finishes. (in addition, the SIGCHLD handler used by that function does not play well with Twisted's own signal handling). For these cases, the twisted.internet.utils.getProcessOutput function can be used. Here is a simple example:

`quotes.py`

```
from twisted.internet import protocol, utils, reactor
from twisted.python import failure
from cStringIO import StringIO

class FortuneQuoter(protocol.Protocol):

    fortune = '/usr/games/fortune'

    def connectionMade(self):
        output = utils.getProcessOutput(self.fortune)
        output.addCallbacks(self.writeResponse, self.noResponse)

    def writeResponse(self, resp):
        self.transport.write(resp)
        self.transport.loseConnection()

    def noResponse(self, err):
        self.transport.loseConnection()


if __name__ == '__main__':
    f = protocol.Factory()
    f.protocol = FortuneQuoter
    reactor.listenTCP(10999, f)
    reactor.run()
```

If you only need the final exit code (like `commands.getstatusoutput(cmd)[0]`), the twisted.internet.utils.getProcessValue function is useful. Here is an example:

`trueandfalse.py`

```
from __future__ import print_function

from twisted.internet import utils, reactor

def printTrueValue(val):
    print("/bin/true exits with rc=%d" % val)
    output = utils.getProcessValue('/bin/false')
    output.addCallback(printFalseValue)

def printFalseValue(val):
    print("/bin/false exits with rc=%d" % val)
    reactor.stop()

output = utils.getProcessValue('/bin/true')
output.addCallback(printTrueValue)
reactor.run()
```

Mapping File Descriptors

“stdin” , “stdout” , and “stderr” are just conventions. Programs which operate as filters generally accept input on fd0, write their output on fd1, and emit error messages on fd2. This is common enough that the standard C library provides macros like “stdin” to mean fd0, and shells interpret the pipe character “|” to mean “redirect fd1 from one command into fd0 of the next command” .

But these are just conventions, and programs are free to use additional file descriptors or even ignore the standard three entirely. The”childFDs” argument allows you to specify exactly what kind of files descriptors the child process should be given.

Each child FD can be put into one of three states:

- Mapped to a parent FD: this causes the child’s reads and writes to come from or go to the same source/destination as the parent.
- Feeding into a pipe which can be read by the parent.
- Feeding from a pipe which the parent writes into.

Mapping the child FDs to the parent’s is very commonly used to send the child’s stderr output to the same place as the parent’s. When you run a program from the shell, it will typically leave fds 0, 1, and 2 mapped to the shell’s 0, 1, and 2, allowing you to see the child program’s output on the same terminal you used to launch the child. Likewise, inetd will typically map both stdin and stdout to the network socket, and may map stderr to the same socket or to some kind of logging mechanism. This allows the child program to be implemented with no knowledge of the network: it merely speaks its protocol by doing reads on fd0 and writes on fd1.

Feeding into a parent’s read pipe is used to gather output from the child, and is by far the most common way of interacting with child processes.

Feeding from a parent’s write pipe allows the parent to control the child. Programs like “bc” or “ftp” can be controlled this way, by writing commands into their stdin stream.

The “childFDs” dictionary maps file descriptor numbers (as will be seen by the child process) to one of these three states. To map the fd to one of the parent’s fds, simply provide the fd number as the value. To map it to a read pipe, use the string “r” as the value. To map it to a write pipe, use the string “w” .

For example, the default mapping sets up the standard stdin/stdout/stderr pipes. It is implemented with the following dictionary:

```
childFDs = { 0: "w", 1: "r", 2: "r" }
```

To launch a process which reads and writes to the same places that the parent python program does, use this:

```
childFDs = { 0: 0, 1: 1, 2: 2}
```

To write into an additional fd (say it is fd number 4), use this:

```
childFDs = { 0: "w", 1: "r", 2: "r" , 4: "w"}
```

ProcessProtocols with extra file descriptors

When you provide a “childFDs” dictionary with more than the normal three fds, you need additional methods to access those pipes. These methods are more generalized than the `.outReceived` ones described above. In fact, those methods (`outReceived` and `errReceived`) are actually just wrappers left in for compatibility with older code, written before this generalized fd mapping was implemented. The new list of things that can happen to your ProcessProtocol is as follows:

- `.connectionMade` : This is called when the program is started.

- `.childDataReceived(childFD, data)` : This is called with data that was received from one of the process' output pipes (i.e. where the childFDs value was "r" . The actual file number (from the point of view of the child process) is in "childFD" . For compatibility, the default implementation of `.childDataReceived` dispatches to `.outReceived` or `.errReceived` when "childFD" is 1 or 2.
- `.childConnectionLost(childFD)` : This is called when the reactor notices that one of the process' pipes has been closed. This either means you have just closed down the parent's end of the pipe (with `.transport.closeChildFD`), the child closed the pipe explicitly (sometimes to indicate EOF), or the child process has terminated and the kernel has closed all of its pipes. The "childFD" argument tells you which pipe was closed. Note that you can only find out about file descriptors which were mapped to pipes: when they are mapped to existing fds the parent has no way to notice when they've been closed. For compatibility, the default implementation dispatches to `.inConnectionLost` , `.outConnectionLost` , or `.errConnectionLost` .
- `.processEnded(status)` : This is called when the child process has been reaped, and all pipes have been closed. This insures that all data written by the child prior to its death will be received before `.processEnded` is invoked.

In addition to those methods, there are other methods available to influence the child process:

- `self.transport.writeToChild(childFD, data)` : Stuff some data into an input pipe. `.write` simply writes to childFD=0.
- `self.transport.closeChildFD(childFD)` : Close one of the child's pipes. Closing an input pipe is a common way to indicate EOF to the child process. Closing an output pipe is neither very friendly nor very useful.

Examples

GnuPG, the encryption program, can use additional file descriptors to accept a passphrase and emit status output. These are distinct from stdin (used to accept the crypttext), stdout (used to emit the plaintext), and stderr (used to emit human-readable status/warning messages). The passphrase FD reads until the pipe is closed and uses the resulting string to unlock the secret key that performs the actual decryption. The status FD emits machine-parseable status messages to indicate the validity of the signature, which key the message was encrypted to, etc.

gpg accepts command-line arguments to specify what these fds are, and then assumes that they have been opened by the parent before the gpg process is started. It simply performs reads and writes to these fd numbers.

To invoke gpg in decryption/verification mode, you would do something like the following:

```
class GPGProtocol(ProcessProtocol):
    def __init__(self, crypttext):
        self.crypttext = crypttext
        self.plaintext = ""
        self.status = ""
    def connectionMade(self):
        self.transport.writeToChild(3, self.passphrase)
        self.transport.closeChildFD(3)
        self.transport.writeToChild(0, self.crypttext)
        self.transport.closeChildFD(0)
    def childDataReceived(self, childFD, data):
        if childFD == 1: self.plaintext += data
        if childFD == 4: self.status += data
    def processEnded(self, status):
        rc = status.value.exitCode
        if rc == 0:
            self.deferred.callback(self)
        else:
```

```
            self.deferred.errback(rc)

def decrypt(crypttext):
    gp = GPGProtocol(crypttext)
    gp.deferred = Deferred()
    cmd = ["gpg", "--decrypt", "--passphrase-fd", "3", "--status-fd", "4",
           "--batch"]
    p = reactor.spawnProcess(gp, cmd[0], cmd, env=None,
                             childFDs={0:"w", 1:"r", 2:2, 3:"w", 4:"r"})
    return gp.deferred
```

In this example, the status output could be parsed after the fact. It could, of course, be parsed on the fly, as it is a simple line-oriented protocol. Methods from LineReceiver could be mixed in to make this parsing more convenient.

The stderr mapping ("2:2") used will cause any GPG errors to be emitted by the parent program, just as if those errors had caused in the parent itself. This is sometimes desirable (it roughly corresponds to letting exceptions propagate upwards), especially if you do not expect to encounter errors in the child process and want them to be more visible to the end user. The alternative is to map stderr to a read-pipe and handle any such output from within the ProcessProtocol (roughly corresponding to catching the exception locally).

Introduction to Deferreds

This document introduces Deferreds, Twisted's preferred mechanism for controlling the flow of asynchronous code. Don't worry if you don't know what that means yet – that's why you are here!

It is intended for newcomers to Twisted, and was written particularly to help people read and understand code that already uses Deferreds.

This document assumes you have a good working knowledge of Python. It assumes no knowledge of Twisted.

By the end of the document, you should understand what Deferreds are and how they can be used to coordinate asynchronous code. In particular, you should be able to:

- Read and understand code that uses Deferreds
- Translate from synchronous code to asynchronous code and back again
- Implement any sort of error-handling for asynchronous code that you wish

The joy of order

When you write Python code, one prevailing, deep, unassailled assumption is that a line of code within a block is only ever executed after the preceding line is finished.

```
pod_bay_doors.open()
pod.launch()
```

The pod bay doors open, and only *then* does the pod launch. That's wonderful. One-line-after-another is a built-in mechanism in the language for encoding the order of execution. It's clear, terse, and unambiguous.

Exceptions make things more complicated. If `pod_bay_doors.open()` raises an exception, then we cannot know with certainty that it completed, and so it would be wrong to proceed blithely to the next line. Thus, Python gives us `try`, `except`, `finally`, and `else`, which together model almost every conceivable way of handling a raised exception, and tend to work really well.

Function application is the other way we encode order of execution:

```
pprint(sorted(x.get_names()))
```

First `x.get_names()` gets called, then `sorted` is called with its return value, and then `pprint` with whatever `sorted` returns.

It can also be written as:

```
names = x.get_names()
sorted_names = sorted(names)
pprint(sorted_names)
```

Sometimes it leads us to encode the order when we don't need to, as in this example:

```
from __future__ import print_function

total = 0
for account in accounts:
    total += account.get_balance()
print("Total balance ${}".format(total))
```

But that's normally not such a big deal.

All in all, things are pretty good, and all of the explanation above is laboring familiar and obvious points. One line comes after another and one thing happens after another, and both facts are inextricably tied.

But what if we had to do it differently?

A hypothetical problem

What if we could no longer rely on the previous line of code being finished (whatever that means) before we started to interpret & execute the next line of code? What if `pod_bay_doors.open()` returned immediately, triggering something somewhere else that would eventually open the pod bay doors, recklessly sending the Python interpreter plunging into `pod.launch()` ?

That is, what would we do if the order of execution did not match the order of lines of Python? If "returning" no longer meant "finishing"?

Asynchronous operations?

How would we prevent our pod from hurtling into the still-closed doors? How could we respond to a potential failure to open the doors at all? What if opening the doors gave us some crucial information that we needed in order to launch the pod? How would we get access to that information?

And, crucially, since we are writing code, how can we write our code so that we can build *other* code on top of it?

The components of a solution

We would still need a way of saying "do *this* only when *that* has finished".

We would need a way of distinguishing between successful completion and interrupted processing, normally modeled with `try`, `expect`, `else`, and `finally`.

We need a mechanism for getting return failures and exception information from the thing that just executed to the thing that needs to happen next.

We need somehow to be able to operate on results that we don't have yet. Instead of acting, we need to make and encode plans for how we would act if we could.

Unless we hack the interpreter somehow, we would need to build this with the Python language constructs we are given: methods, functions, objects, and the like.

Perhaps we want something that looks a little like this:

```
placeholder = pod_bay_doors.open()
placeholder.when_done(pod.launch)
```

One solution: Deferred

Twisted tackles this problem with Deferreds, a type of object designed to do one thing, and one thing only: encode an order of execution separately from the order of lines in Python source code.

It doesn't deal with threads, parallelism, signals, or subprocesses. It doesn't know anything about an event loop, greenlets, or scheduling. All it knows about is what order to do things in. How does it know that? Because we explicitly tell it the order that we want.

Thus, instead of writing:

```
pod_bay_doors.open()
pod.launch()
```

We write:

```
d = pod_bay_doors.open()
d.addCallback(lambda ignored: pod.launch())
```

That introduced a dozen new concepts in a couple of lines of code, so let's break it down. If you think you've got it, you might want to skip to the next section.

Here, `pod_bay_doors.open()` is returning a Deferred, which we assign to `d`. We can think of `d` as a placeholder, representing the value that `open()` will eventually return when it finally gets around to finishing.

To "do this next", we add a *callback* to `d`. A callback is a function that will be called with whatever `open()` eventually returns. In this case, we don't care, so we make a function with a single, ignored parameter that just calls `pod.launch()`.

So, we've replaced the "order of lines is order of execution" with a deliberate, in-Python encoding of the order of execution, where `d` represents the particular flow and `d.addCallback` replaces "new line".

Of course, programs generally consist of more than two lines, and we still don't know how to deal with failure.

Getting it right: The failure cases

In what follows, we are going to take each way of expressing order of operations in normal Python (using lines of code and `try`/`except`) and translate them into an equivalent code built with Deferred objects.

This is going to be a bit painstaking, but if you want to really understand how to use Deferreds and maintain code that uses them, it is worth understanding each example below.

One thing, then another, then another

Recall our example from earlier:

```
pprint(sorted(x.get_names()))
```

Also written as:

```
names = x.get_names()
sorted_names = sorted(names)
pprint(sorted_names)
```

What if neither `get_names` nor `sorted` can be relied on to finish before they return? That is, if both are asynchronous operations?

Well, in Twisted-speak they would return Deferreds and so we would write:

```
d = x.get_names()
d.addCallback(sorted)
d.addCallback(pprint)
```

Eventually, `sorted` will get called with whatever `get_names` finally delivers. When `sorted` finishes, `pprint` will be called with whatever it delivers.

We could also write this as:

```
x.get_names().addCallback(sorted).addCallback(pprint)
```

Since `d.addCallback` returns `d`.

Simple failure handling

We often want to write code equivalent to this:

```
try:
    x.get_names()
except Exception as e:
    report_error(e)
```

How would we write this with Deferreds?

```
d = x.get_names()
d.addErrback(report_error)
```

errback is the Twisted name for a callback that is called when an error is received.

This glosses over an important detail. Instead of getting the exception object `e`, `report_error` would get a Failure object, which has all of the useful information that `e` does, but is optimized for use with Deferreds.

We'll dig into that a bit later, after we've dealt with all of the other combinations of exceptions.

Handle an error, but do something else on success

What if we want to do something after our `try` block if it actually worked? Abandoning our contrived examples and reaching for generic variable names, we get:

```
try:
    y = f()
except Exception as e:
    g(e)
else:
    h(y)
```

Well, we'd write it like this with Deferreds:

```
d = f()
d.addCallbacks(h, g)
```

Where `addCallbacks` means "add a callback and an errback at the same time". `h` is the callback, `g` is the errback.

Now that we have `addCallbacks` along with `addErrback` and `addCallback`, we can match any possible combination of `try`, `except`, `else`, and `finally` by varying the order in which we call them. Explaining exactly how it works is tricky (although the *Deferred reference* does rather a good job), but once we're through all of the examples it ought to be clearer.

Handle an error, then proceed anyway

What if we want to do something after our `try`/`except` block, regardless of whether or not there was an exception? That is, what if we wanted to do the equivalent of this generic code:

```
try:
    y = f()
except Exception as e:
    y = g(e)
h(y)
```

And with Deferreds:

```
d = f()
d.addErrback(g)
d.addCallback(h)
```

Because `addErrback` returns `d`, we can chain the calls like so:

```
f().addErrback(g).addCallback(h)
```

The order of `addErrback` and `addCallback` matters. In the next section, we can see what would happen when we swap them around.

Handle an error for the entire operation

What if we want to wrap up a multi-step operation in one exception handler?

```
try:
    y = f()
    z = h(y)
except Exception as e:
    g(e)
```

With Deferreds, it would look like this:

```
d = f()
d.addCallback(h)
d.addErrback(g)
```

Or, more succinctly:

```
d = f().addCallback(h).addErrback(g)
```

Do something regardless

What about `finally`? How do we do something regardless of whether or not there was an exception? How do we translate this:

```
try:
    y = f()
finally:
    g()
```

Well, roughly we do this:

```
d = f()
d.addBoth(g)
```

This adds `g` as both the callback and the errback. It is equivalent to:

```
d.addCallbacks(g, g)
```

Why "roughly"? Because if `f` raises, `g` will be passed a Failure object representing the exception. Otherwise, `g` will be passed the asynchronous equivalent of the return value of `f()` (i.e. `y`).

Inline callbacks - using 'yield'

Twisted features a decorator named `inlineCallbacks` which allows you to work with Deferreds without writing callback functions.

This is done by writing your code as generators, which *yield* `Deferred`s instead of attaching callbacks.

Consider the following function written in the traditional `Deferred` style:

```
def getUsers():
   d = makeRequest("GET", "/users")
   d.addCallback(json.loads)
   return d
```

using `inlineCallbacks`, we can write this as:

```
from twisted.internet.defer import inlineCallbacks, returnValue

@inlineCallbacks
def getUsers(self):
    responseBody = yield makeRequest("GET", "/users")
    returnValue(json.loads(responseBody))
```

a couple of things are happening here:

1. instead of calling `addCallback` on the `Deferred` returned by `makeRequest`, we *yield* it. This causes Twisted to return the `Deferred`'s result to us.
2. we use `returnValue` to propagate the final result of our function. Because this function is a generator, we cannot use the return statement; that would be a syntax error.

Note: New in version 15.0.

On Python 3.3 and above, instead of writing `returnValue(json.loads(responseBody))` you can instead write `return json.loads(responseBody)`. This can be a significant readability advantage, but unfortunately if you need compatibility with Python 2, this isn't an option.

Both versions of `getUsers` present exactly the same API to their callers: both return a `Deferred` that fires with the parsed JSON body of the request. Though the `inlineCallbacks` version looks like synchronous code, which blocks while waiting for the request to finish, each `yield` statement allows other code to run while waiting for the `Deferred` being yielded to fire.

`inlineCallbacks` become even more powerful when dealing with complex control flow and error handling. For example, what if `makeRequest` fails due to a connection error? For the sake of this example, let's say we want to log the exception and return an empty list.

```
def getUsers():
   d = makeRequest("GET", "/users")

   def connectionError(failure):
       failure.trap(ConnectionError)
       log.failure("makeRequest failed due to connection error",
                   failure)
       return []

   d.addCallbacks(json.loads, connectionError)
   return d
```

With `inlineCallbacks`, we can rewrite this as:

```
@inlineCallbacks
def getUsers(self):
    try:
        responseBody = yield makeRequest("GET", "/users")
    except ConnectionError:
       log.failure("makeRequest failed due to connection error")
       returnValue([])

    returnValue(json.loads(responseBody))
```

Our exception handling is simplified because we can use Python's familiar `try` / `except` syntax for handling `ConnectionErrors`.

Coroutines with async/await

Note: Only available on Python 3.5 and higher.

New in version 16.4.

On Python 3.5 and higher, the **PEP 492** ("Coroutines with async and await syntax") "await" functionality can be used with Deferreds by the use of ensureDeferred. It is similar to `inlineCallbacks`, except that it uses the `await` keyword instead of `yield`, the `return` keyword instead of `returnValue`, and is a function rather than a decorator.

Calling a coroutine (that is, the result of a function defined by `async def funcname():` with ensureDeferred will allow you to "await" on Deferreds inside it, and will return a standard Deferred. You can mix and match code which uses regular Deferreds, `inlineCallbacks`, and `ensureDeferred` freely.

Awaiting on a Deferred which fires with a Failure will raise the exception inside your coroutine as if it were regular Python. If your coroutine raises an exception, it will be translated into a Failure fired on the Deferred that `ensureDeferred` returns for you. Calling `return` will cause the Deferred that `ensureDeferred` returned for you to fire with a result.

```
import json
from twisted.internet.defer import ensureDeferred
from twisted.logger import Logger
log = Logger()

async def getUsers():
    try:
        return json.loads(await makeRequest("GET", "/users"))
    except ConnectionError:
        log.failure("makeRequest failed due to connection error")
        return []

def do():
    d = ensureDeferred(getUsers())
    d.addCallback(print)
    return d
```

When writing coroutines, you do not need to use ensureDeferred when you are writing a coroutine which calls other coroutines which await on Deferreds; you can just `await` on it directly. For example:

```
async def foo():
    res = await someFunctionThatReturnsADeferred()
    return res

async def bar():
    baz = await someOtherDeferredFunction()
    fooResult = await foo()
    return baz + fooResult

def myDeferredReturningFunction():
    coro = bar()
    return ensureDeferred(coro)
```

Even though Deferreds were used in both coroutines, only `bar` had to be wrapped in ensureDeferred to return a Deferred.

Conclusion

You have been introduced to asynchronous code and have seen how to use Deferreds to:

- Do something after an asynchronous operation completes successfully
- Use the result of a successful asynchronous operation
- Catch errors in asynchronous operations
- Do one thing if an operation succeeds, and a different thing if it fails
- Do something after an error has been handled successfully
- Wrap multiple asynchronous operations with one error handler

- Do something after an asynchronous operation, regardless of whether it succeeded or failed
- Write code without callbacks using `inlineCallbacks`
- Write coroutines that interact with Deferreds using `ensureDeferred`

These are very basic uses of Deferred. For detailed information about how they work, how to combine multiple Deferreds, and how to write code that mixes synchronous and asynchronous APIs, see the *Deferred reference*. Alternatively, read about how to write functions that *generate Deferreds*.

Deferred Reference

This document is a guide to the behaviour of the twisted.internet.defer.Deferred object, and to various ways you can use them when they are returned by functions.

This document assumes that you are familiar with the basic principle that the Twisted framework is structured around: asynchronous, callback-based programming, where instead of having blocking code in your program or using threads to run blocking code, you have functions that return immediately and then begin a callback chain when data is available.

After reading this document, the reader should expect to be able to deal with most simple APIs in Twisted and Twisted-using code that return Deferreds.

- what sorts of things you can do when you get a Deferred from a function call; and
- how you can write your code to robustly handle errors in Deferred code.

Deferreds

Twisted uses the Deferred object to manage the callback sequence. The client application attaches a series of functions to the deferred to be called in order when the results of the asynchronous request are available (this series of functions is known as a series of **callbacks**, or a **callback chain**), together with a series of functions to be called if there is an error in the asynchronous request (known as a series of **errbacks** or an **errback chain**). The asynchronous library code calls the first callback when the result is available, or the first errback when an error occurs, and the `Deferred` object then hands the results of each callback or errback function to the next function in the chain.

Callbacks

A twisted.internet.defer.Deferred is a promise that a function will at some point have a result. We can attach callback functions to a Deferred, and once it gets a result these callbacks will be called. In addition Deferreds allow the developer to register a callback for an error, with the default behavior of logging the error. The deferred mechanism standardizes the application programmer's interface with all sorts of blocking or delayed operations.

```
from twisted.internet import reactor, defer

def getDummyData(inputData):
    """
    This function is a dummy which simulates a delayed result and
    returns a Deferred which will fire with that result. Don't try too
    hard to understand this.
    """
    print('getDummyData called')
    deferred = defer.Deferred()
    # simulate a delayed result by asking the reactor to fire the
    # Deferred in 2 seconds time with the result inputData * 3
    reactor.callLater(2, deferred.callback, inputData * 3)
    return deferred
```

```
def cbPrintData(result):
    """
    Data handling function to be added as a callback: handles the
    data by printing the result
    """
    print('Result received: {}'.format(result))

deferred = getDummyData(3)
deferred.addCallback(cbPrintData)

# manually set up the end of the process by asking the reactor to
# stop itself in 4 seconds time
reactor.callLater(4, reactor.stop)
# start up the Twisted reactor (event loop handler) manually
print('Starting the reactor')
reactor.run()
```

Multiple callbacks

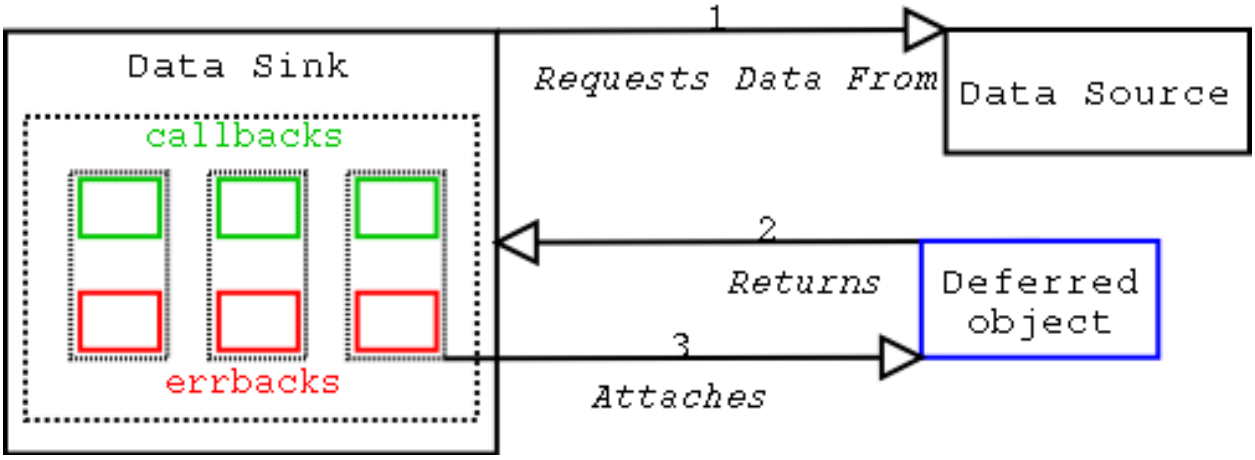
Multiple callbacks can be added to a Deferred. The first callback in the Deferred's callback chain will be called with the result, the second with the result of the first callback, and so on. Why do we need this? Well, consider a Deferred returned by twisted.enterprise.adbapi - the result of a SQL query. A web widget might add a callback that converts this result into HTML, and pass the Deferred onwards, where the callback will be used by twisted to return the result to the HTTP client. The callback chain will be bypassed in case of errors or exceptions.

```
from twisted.internet import reactor, defer

class Getter:
    def gotResults(self, x):
        """
        The Deferred mechanism provides a mechanism to signal error
        conditions.  In this case, odd numbers are bad.

        This function demonstrates a more complex way of starting
        the callback chain by checking for expected results and
        choosing whether to fire the callback or errback chain
        """
        if self.d is None:
            print("Nowhere to put results")
            return

        d = self.d
        self.d = None
        if x % 2 == 0:
            d.callback(x*3)
        else:
            d.errback(ValueError("You used an odd number!"))

    def _toHTML(self, r):
        """
        This function converts r to HTML.

        It is added to the callback chain by getDummyData in
        order to demonstrate how a callback passes its own result
        to the next callback
```

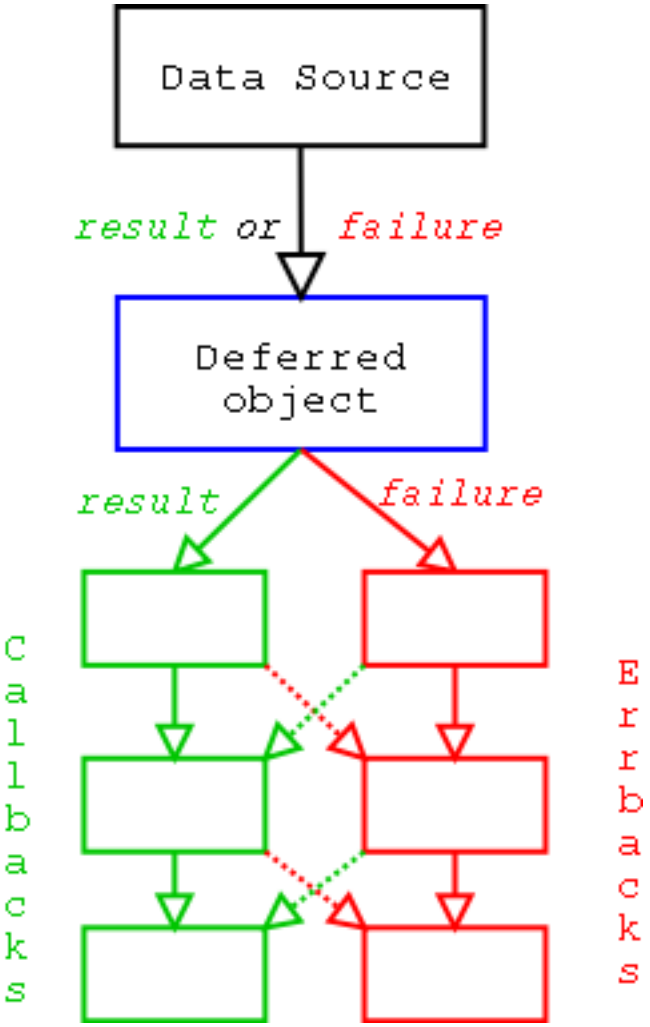
```
        """
        return "Result: %s" % r

    def getDummyData(self, x):
        """
        The Deferred mechanism allows for chained callbacks.
        In this example, the output of gotResults is first
        passed through _toHTML on its way to printData.

        Again this function is a dummy, simulating a delayed result
        using callLater, rather than using a real asynchronous
        setup.
        """
        self.d = defer.Deferred()
        # simulate a delayed result by asking the reactor to schedule
        # gotResults in 2 seconds time
        reactor.callLater(2, self.gotResults, x)
        self.d.addCallback(self._toHTML)
        return self.d

def cbPrintData(result):
    print(result)

def ebPrintError(failure):
    import sys
    sys.stderr.write(str(failure))

# this series of callbacks and errbacks will print an error message
g = Getter()
d = g.getDummyData(3)
d.addCallback(cbPrintData)
d.addErrback(ebPrintError)

# this series of callbacks and errbacks will print "Result: 12"
g = Getter()
d = g.getDummyData(4)
d.addCallback(cbPrintData)
d.addErrback(ebPrintError)

reactor.callLater(4, reactor.stop)
reactor.run()
```

Note: Pay particular attention to the handling of `self.d` in the `gotResults` method. Before the `Deferred` is fired with a result or an error, the attribute is set to `None` so that the `Getter` instance no longer has a reference to the `Deferred` about to be fired. This has several benefits. First, it avoids any chance `Getter.gotResults` will accidentally fire the same `Deferred` more than once (which would result in an `AlreadyCalledError` exception). Second, it allows a callback on that `Deferred` to call `Getter.getDummyData` (which sets a new value for the `d` attribute) without causing problems. Third, it makes the Python garbage collector's job easier by eliminating a reference cycle.

Visual Explanation



1. Requesting method (data sink) requests data, gets Deferred object.
2. Requesting method attaches callbacks to Deferred object.



1. When the result is ready, give it to the Deferred object. `.callback(result)` if the operation succeeded, `.errback(failure)` if it failed. Note that `failure` is typically an instance of a twisted.python.failure.Failure instance.
2. Deferred object triggers previously-added (call/err)back with the `result` or `failure`. Execution then follows the following rules, going down the chain of callbacks to be processed.
   - Result of the callback is always passed as the first argument to the next callback, creating a chain of processors.
   - If a callback raises an exception, switch to errback.
   - An unhandled failure gets passed down the line of errbacks, this creating an asynchronous analog to a series to a series of `except:` statements.
   - If an errback doesn't raise an exception or return a twisted.python.failure.Failure instance, switch to callback.

Errbacks

Deferred's error handling is modeled after Python's exception handling. In the case that no errors occur, all the callbacks run, one after the other, as described above.

If the errback is called instead of the callback (e.g. because a DB query raised an error), then a twisted.python.failure.Failure is passed into the first errback (you can add multiple errbacks, just like with callbacks). You can think of your errbacks as being like `except` blocks of ordinary Python code.

Unless you explicitly `raise` an error in an except block, the `Exception` is caught and stops propagating, and normal execution continues. The same thing happens with errbacks: unless you explicitly `return` a `Failure` or (re-)raise an exception, the error stops propagating, and normal callbacks continue executing from that point (using the value returned from the errback). If the errback does return a `Failure` or raise an exception, then that is passed to the next errback, and so on.

Note: If an errback doesn't return anything, then it effectively returns `None`, meaning that callbacks will continue to be executed after this errback. This may not be what you expect to happen, so be careful. Make sure your errbacks return a `Failure` (probably the one that was passed to it), or a meaningful return value for the next callback.

Also, twisted.python.failure.Failure instances have a useful method called trap, allowing you to effectively do the equivalent of:

```
try:
    # code that may throw an exception
    cookSpamAndEggs()
except (SpamException, EggException):
    # Handle SpamExceptions and EggExceptions
    ...
```

You do this by:

```
def errorHandler(failure):
    failure.trap(SpamException, EggException)
    # Handle SpamExceptions and EggExceptions

d.addCallback(cookSpamAndEggs)
d.addErrback(errorHandler)
```

If none of arguments passed to `failure.trap` match the error encapsulated in that `Failure`, then it re-raises the error.

There's another potential "gotcha" here. There's a method twisted.internet.defer.Deferred.addCallbacks which is similar to, but not exactly the same as, `addCallback` followed by `addErrback`. In particular, consider these two cases:

```
# Case 1
d = getDeferredFromSomewhere()
d.addCallback(callback1)        # A
d.addErrback(errback1)          # B
d.addCallback(callback2)
d.addErrback(errback2)

# Case 2
d = getDeferredFromSomewhere()
d.addCallbacks(callback1, errback1)   # C
d.addCallbacks(callback2, errback2)
```

If an error occurs in `callback1`, then for Case 1 `errback1` will be called with the failure. For Case 2, `errback2` will be called. Be careful with your callbacks and errbacks.

What this means in a practical sense is in Case 1, the callback in line A will handle a success condition from `getDeferredFromSomewhere`, and the errback in line B will handle any errors that occur *from either the upstream source, or that occur in A*. In Case 2, the errback in line C *will only handle an error condition raised by* `getDeferredFromSomewhere`, it will not do any handling of errors raised in `callback1`.

Unhandled Errors

If a Deferred is garbage-collected with an unhandled error (i.e. it would call the next errback if there was one), then Twisted will write the error's traceback to the log file. This means that you can typically get away with not adding errbacks and still get errors logged. Be careful though; if you keep a reference to the Deferred around, preventing it from being garbage-collected, then you may never see the error (and your callbacks will mysteriously seem to have never been called). If unsure, you should explicitly add an errback after your callbacks, even if all you do is:

```
# Make sure errors get logged
from twisted.python import log
d.addErrback(log.err)
```

Handling either synchronous or asynchronous results

In some applications, there are functions that might be either asynchronous or synchronous. For example, a user authentication function might be able to check in memory whether a user is authenticated, allowing the authentication function to return an immediate result, or it may need to wait on network data, in which case it should return a Deferred to be fired when that data arrives. However, a function that wants to check if a user is authenticated will then need to accept both immediate results *and* Deferreds.

In this example, the library function `authenticateUser` uses the application function `isValidUser` to authenticate a user:

```
def authenticateUser(isValidUser, user):
    if isValidUser(user):
        print("User is authenticated")
    else:
        print("User is not authenticated")
```

However, it assumes that `isValidUser` returns immediately, whereas `isValidUser` may actually authenticate the user asynchronously and return a Deferred. It is possible to adapt this trivial user authentication code to ac-

cept either a synchronous `isValidUser` or an asynchronous `isValidUser`, allowing the library to handle either type of function. It is, however, also possible to adapt synchronous functions to return Deferreds. This section describes both alternatives: handling functions that might be synchronous or asynchronous in the library function (`authenticateUser`) or in the application code.

Handling possible Deferreds in the library code

Here is an example of a synchronous user authentication function that might be passed to `authenticateUser`:

`synch-validation.py`

```
def synchronousIsValidUser(user):
    '''
    Return true if user is a valid user, false otherwise
    '''
    return user in ["Alice", "Angus", "Agnes"]
```

However, here's an `asynchronousIsValidUser` function that returns a Deferred:

```
from twisted.internet import reactor, defer

def asynchronousIsValidUser(user):
    d = defer.Deferred()
    reactor.callLater(2, d.callback, user in ["Alice", "Angus", "Agnes"])
    return d
```

Our original implementation of `authenticateUser` expected `isValidUser` to be synchronous, but now we need to change it to handle both synchronous and asynchronous implementations of `isValidUser`. For this, we use maybeDeferred to call `isValidUser`, ensuring that the result of `isValidUser` is a Deferred, even if `isValidUser` is a synchronous function:

```
from twisted.internet import defer

def printResult(result):
    if result:
        print("User is authenticated")
    else:
        print("User is not authenticated")

def authenticateUser(isValidUser, user):
    d = defer.maybeDeferred(isValidUser, user)
    d.addCallback(printResult)
```

Now `isValidUser` could be either `synchronousIsValidUser` or `asynchronousIsValidUser`.

It is also possible to modify `synchronousIsValidUser` to return a Deferred, see *Generating Deferreds* for more information.

Cancellation

Motivation

A Deferred may take any amount of time to be called back; in fact, it may never be called back. Your users may not be that patient. Since all actions taken when the Deferred completes are in your application or library's callback code, you always have the option of simply disregarding the result when you receive it, if it's been too long. However, while

you're ignoring it, the underlying operation represented by that Deferred is still chugging along in the background, possibly consuming resources such as CPU time, memory, network bandwidth and maybe even disk space. So, when the user has closed the window, hit the cancel button, disconnected from your server or sent a "stop" network message, you will want to announce your indifference to the result of that operation so that the originator of the Deferred can clean everything up and free those resources to be put to better use.

Cancellation for Applications which Consume Deferreds

Here's a simple example. You're connecting to an external host with an *endpoint*, but that host is really slow. You want to put a "cancel" button into your application to terminate the connection attempt, so the user can try connecting to a different host instead. Here's a simple sketch of such an application, with the actual user interface left as an exercise for the reader:

```
def startConnecting(someEndpoint):
    def connected(it):
        "Do something useful when connected."
    return someEndpoint.connect(myFactory).addCallback(connected)
# ...
connectionAttempt = startConnecting(endpoint)
def cancelClicked():
    connectionAttempt.cancel()
```

Obviously (I hope), startConnecting is meant to be called by some UI element that lets the user choose what host to connect to and then constructs an appropriate endpoint (perhaps using `twisted.internet.endpoints.clientFromString`). Then, a cancel button, or similar, is hooked up to the `cancelClicked`.

When `connectionAttempt.cancel` is invoked, that will:

1. cause the underlying connection operation to be terminated, if it is still ongoing
2. cause the connectionAttempt Deferred to be completed, one way or another, in a timely manner
3. *likely* cause the connectionAttempt Deferred to be errbacked with CancelledError

You may notice that that set of consequences is very heavily qualified. Although cancellation indicates the calling API's *desire* for the underlying operation to be stopped, the underlying operation cannot necessarily react immediately. Even in this very simple example, there is already one thing that might not be interruptible: platform-native name resolution blocks, and therefore needs to be executed in a thread; the connection operation can't be cancelled if it's stuck waiting for a name to be resolved in this manner. So, the Deferred that you are cancelling may not callback or errback right away.

A Deferred may wait upon another Deferred at any point in its callback chain (see "Handling...asynchronous results", above). There's no way for a particular point in the callback chain to know if everything is finished. Since multiple layers of the callback chain may wish to cancel the same Deferred, any layer may call `.cancel()` at any time. The `.cancel()` method never raises any exception or returns any value; you may call it repeatedly, even on a Deferred which has already fired, or which has no remaining callbacks. The main reason for all these qualifications, aside from specific examples, is that anyone who instantiates a Deferred may supply it with a cancellation function; that function can do absolutely anything that it wants to. Ideally, anything it does will be in the service of stopping the operation your requested, but there's no way to guarantee any exact behavior across all Deferreds that might be cancelled. Cancellation of Deferreds is best effort. This may be the case for a number of reasons:

1. The `Deferred` doesn't know how to cancel the underlying operation.
2. The underlying operation may have reached an uncancellable state, because some irreversible operation has been done.
3. The `Deferred` may already have a result, and so there's nothing to cancel.

Calling `cancel()` will always succeed without an error regardless of whether or not cancellation was possible. In cases 1 and 2 the `Deferred` may well errback with a `twisted.internet.defer.CancelledError` while the underlying operation continues. `Deferred` s that support cancellation should document what they do when cancelled, if they are uncancellable in certain edge cases, etc..

If the cancelled `Deferred` is waiting on another `Deferred`, the cancellation will be forwarded to the other `Deferred`.

Default Cancellation Behavior

All Deferreds support cancellation. However, by default, they support a very rudimentary form of cancellation which doesn't free any resources.

Consider this example of a Deferred which is ignorant of cancellation:

```
operation = Deferred()
def x(result):
    print("Hooray, a result:" + repr(x))
operation.addCallback(x)
# ...
def operationDone():
    operation.callback("completed")
```

A caller of an API that receives `operation` may call `cancel` on it. Since `operation` does not have a cancellation function, one of two things will happen.

1. If `operationDone` has been called, and the operation has completed, nothing much will change. `operation` will still have a result, and there are no more callbacks, so there's no observable change in behavior.
2. If `operationDone` has *not* yet been invoked, then `operation` will be immediately errbacked with a `CancelledError`.

   However, once it's cancelled, there's no way to tell `operationDone` not to run; it will eventually call `operation.callback` later. In normal operation, issuing `callback` on a `Deferred` that has already called back results in an `AlreadyCalledError`, and this would cause an ugly traceback that could not be caught. Therefore, `.callback` can be invoked exactly once, causing a no-op, on a `Deferred` which has been cancelled but has no canceller. If you call it multiple times, you will still get an `AlreadyCalledError` exception.

Creating Cancellable Deferreds: Custom Cancellation Functions

Let's imagine you are implementing an HTTP client, which returns a Deferred firing with the response from the server. Cancellation is best achieved by closing the connection. In order to make cancellation do that, all you have to do is pass a function to the constructor of the Deferred (it will get called with the Deferred that is being cancelled):

```
class HTTPClient(Protocol):
    def request(self, method, path):
        self.resultDeferred = Deferred(
            lambda ignore: self.transport.abortConnection())
        request = b"%s %s HTTP/1.0\r\n\r\n" % (method, path)
        self.transport.write(request)
        return self.resultDeferred

    def dataReceived(self, data):
```

```
        # ... parse HTTP response ...
        # ... eventually call self.resultDeferred.callback() ...
```

Now if someone calls `cancel()` on the `Deferred` returned from `HTTPClient.request()`, the HTTP request will be cancelled (assuming it's not too late to do so). Care should be taken not to `callback()` a Deferred that has already been cancelled.

Timeouts

Timeouts are a special case of *Cancellation*. Let's say we have a Deferred representing a task that may take a long time. We want to put an upper bound on that task, so we want the Deferred to time out X seconds in the future.

A convenient API to do so is Deferred.addTimeout. By default, it will fail with a TimeoutError if the Deferred hasn't fired (with either an errback or a callback) within `timeout` seconds.

```
import random
from twisted.internet import task

def f():
    return "Hopefully this will be called in 3 seconds or less"

def main(reactor):
    delay = random.uniform(1, 5)

    def called(result):
        print("{0} seconds later:".format(delay), result)

    d = task.deferLater(reactor, delay, f)
    d.addTimeout(3, reactor).addBoth(called)

    return d

# f() will be timed out if the random delay is greater than 3 seconds
task.react(main)
```

Deferred.addTimeout uses the Deferred.cancel function under the hood, but can distinguish between a user's call to Deferred.cancel and a cancellation due to a timeout. By default, Deferred.addTimeout translates a CancelledError produced by the timeout into a TimeoutError.

However, if you provided a custom *cancellation* when creating the Deferred, then cancelling it may not produce a CancelledError. In this case, the default behavior of Deferred.addTimeout is to preserve whatever callback or errback value your custom cancellation function produced. This can be useful if, for instance, a cancellation or timeout should produce a default value instead of an error.

Deferred.addTimeout also takes an optional callable `onTimeoutCancel` which is called immediately after the deferred times out. `onTimeoutCancel` is not called if it the deferred is otherwise cancelled before the timeout. It takes an arbitrary value, which is the value of the deferred at that exact time (probably a CancelledError Failure), and the `timeout`. This can be useful if, for instance, the cancellation or timeout does not result in an error but you want to log the timeout anyway. It can also be used to alter the return value.

```
from twisted.internet import task, defer

def logTimeout(result, timeout):
    print("Got {0!r} but actually timed out after {1} seconds".format(
        result, timeout))
    return result + " (timed out)"
```

```
def main(reactor):
    # generate a deferred with a custom canceller function, and never
    # never callback or errback it to guarantee it gets timed out
    d = defer.Deferred(lambda c: c.callback("Everything's ok!"))
    d.addTimeout(2, reactor, onTimeoutCancel=logTimeout)
    d.addBoth(print)
    return d

task.react(main)
```

Note that the exact place in the callback chain that Deferred.addTimeout is added determines how much of the callback chain should be timed out. The timeout encompasses all the callbacks and errbacks added to the Deferred before the call to addTimeout, and none of the callbacks and errbacks added after the call. The timeout also starts counting down as soon as soon as it's invoked.

DeferredList

Sometimes you want to be notified after several different events have all happened, rather than waiting for each one individually. For example, you may want to wait for all the connections in a list to close. twisted.internet.defer.DeferredList is the way to do this.

To create a DeferredList from multiple Deferreds, you simply pass a list of the Deferreds you want it to wait for:

```
# Creates a DeferredList
dl = defer.DeferredList([deferred1, deferred2, deferred3])
```

You can now treat the DeferredList like an ordinary Deferred; you can call `addCallbacks` and so on. The DeferredList will call its callback when all the deferreds have completed. The callback will be called with a list of the results of the Deferreds it contains, like so:

```
# A callback that unpacks and prints the results of a DeferredList
def printResult(result):
    for (success, value) in result:
        if success:
            print('Success:', value)
        else:
            print('Failure:', value.getErrorMessage())

# Create three deferreds.
deferred1 = defer.Deferred()
deferred2 = defer.Deferred()
deferred3 = defer.Deferred()

# Pack them into a DeferredList
dl = defer.DeferredList([deferred1, deferred2, deferred3], consumeErrors=True)

# Add our callback
dl.addCallback(printResult)

# Fire our three deferreds with various values.
deferred1.callback('one')
deferred2.errback(Exception('bang!'))
deferred3.callback('three')

# At this point, dl will fire its callback, printing:
#    Success: one
#    Failure: bang!
```

```
#    Success: three
# (note that defer.SUCCESS == True, and defer.FAILURE == False)
```

A standard DeferredList will never call errback, but failures in Deferreds passed to a DeferredList will still errback unless `consumeErrors` is passed `True`. See below for more details about this and other flags which modify the behavior of DeferredList.

Note: If you want to apply callbacks to the individual Deferreds that go into the DeferredList, you should be careful about when those callbacks are added. The act of adding a Deferred to a DeferredList inserts a callback into that Deferred (when that callback is run, it checks to see if the DeferredList has been completed yet). The important thing to remember is that it is *this callback* which records the value that goes into the result list handed to the DeferredList's callback.

Therefore, if you add a callback to the Deferred *after* adding the Deferred to the DeferredList, the value returned by that callback will not be given to the DeferredList's callback. To avoid confusion, we recommend not adding callbacks to a Deferred once it has been used in a DeferredList.

```
def printResult(result):
    print(result)

def addTen(result):
    return result + " ten"

# Deferred gets callback before DeferredList is created
deferred1 = defer.Deferred()
deferred2 = defer.Deferred()
deferred1.addCallback(addTen)
dl = defer.DeferredList([deferred1, deferred2])
dl.addCallback(printResult)
deferred1.callback("one") # fires addTen, checks DeferredList, stores "one ten"
deferred2.callback("two")
# At this point, dl will fire its callback, printing:
#     [(1, 'one ten'), (1, 'two')]

# Deferred gets callback after DeferredList is created
deferred1 = defer.Deferred()
deferred2 = defer.Deferred()
dl = defer.DeferredList([deferred1, deferred2])
deferred1.addCallback(addTen) # will run *after* DeferredList gets its value
dl.addCallback(printResult)
deferred1.callback("one") # checks DeferredList, stores "one", fires addTen
deferred2.callback("two")
# At this point, dl will fire its callback, printing:
#     [(1, 'one'), (1, 'two')]
```

Other behaviours

DeferredList accepts three keyword arguments that modify its behaviour: `fireOnOneCallback`, `fireOnOneErrback` and `consumeErrors`. If `fireOnOneCallback` is set, the DeferredList will immediately call its callback as soon as any of its Deferreds call their callback. Similarly, `fireOnOneErrback` will call errback as soon as any of the Deferreds call their errback. Note that DeferredList is still one-shot, like ordinary Deferreds, so after a callback or errback has been called the DeferredList will do nothing further (it will just silently ignore any other results from its Deferreds).

The `fireOnOneErrback` option is particularly useful when you want to wait for all the results if everything succeeds, but also want to know immediately if something fails.

The `consumeErrors` argument will stop the DeferredList from propagating any errors along the callback chains of any Deferreds it contains (usually creating a DeferredList has no effect on the results passed along the callbacks and errbacks of their Deferreds). Stopping errors at the DeferredList with this option will prevent "Unhandled error in Deferred" warnings from the Deferreds it contains without needing to add extra errbacks[1] . Passing a true value for the `consumeErrors` parameter will not change the behavior of `fireOnOneCallback` or `fireOnOneErrback`.

gatherResults

A common use for DeferredList is to "join" a number of parallel asynchronous operations, finishing successfully if all of the operations were successful, or failing if any one of the operations fails. In this case, twisted.internet.defer.gatherResults is a useful shortcut:

```
from twisted.internet import defer

d1 = defer.Deferred()
d2 = defer.Deferred()
d = defer.gatherResults([d1, d2], consumeErrors=True)

def cbPrintResult(result):
    print(result)

d.addCallback(cbPrintResult)

d1.callback("one")
# nothing is printed yet; d is still awaiting completion of d2
d2.callback("two")
# printResult prints ["one", "two"]
```

The `consumeErrors` argument has the same meaning as it does for *DeferredList*: if true, it causes `gatherResults` to consume any errors in the passed-in Deferreds. Always use this argument unless you are adding further callbacks or errbacks to the passed-in Deferreds, or unless you know that they will not fail. Otherwise, a failure will result in an unhandled error being logged by Twisted. This argument is available since Twisted 11.1.0.

Class Overview

This is an overview API reference for Deferred from the point of using a Deferred returned by a function. It is not meant to be a substitute for the docstrings in the Deferred class, but can provide guidelines for its use.

There is a parallel overview of functions used by the Deferred's *creator* in *Generating Deferreds*.

Basic Callback Functions

- `addCallbacks(self, callback[, errback, callbackArgs, callbackKeywords, errbackArgs, errbackKeywords])`

  This is the method you will use to interact with Deferred. It adds a pair of callbacks "parallel" to each other (see diagram above) in the list of callbacks made when the Deferred is called back to. The signature of a method added using addCallbacks should be `myMethod(result, *methodAsrgs, **methodKeywords)`.

[1] Unless of course a later callback starts a fresh error — but as we've already noted, adding callbacks to a Deferred after its used in a DeferredList is confusing and usually avoided.

If your method is passed in the callback slot, for example, all arguments in the tuple `callbackArgs` will be passed as `*methodArgs` to your method.

There are various convenience methods that are derivative of addCallbacks. I will not cover them in detail here, but it is important to know about them in order to create concise code.

- `addCallback(callback, *callbackArgs, **callbackKeywords)`

  Adds your callback at the next point in the processing chain, while adding an errback that will re-raise its first argument, not affecting further processing in the error case.

  Note that, while addCallbacks (plural) requires the arguments to be passed in a tuple, addCallback (singular) takes all its remaining arguments as things to be passed to the callback function. The reason is obvious: addCallbacks (plural) cannot tell whether the arguments are meant for the callback or the errback, so they must be specifically marked by putting them into a tuple. addCallback (singular) knows that everything is destined to go to the callback, so it can use Python's "*" and "**" syntax to collect the remaining arguments.

- `addErrback(errback, *errbackArgs, **errbackKeywords)`

  Adds your errback at the next point in the processing chain, while adding a callback that will return its first argument, not affecting further processing in the success case.

- `addBoth(callbackOrErrback, *callbackOrErrbackArgs, **callbackOrErrbackKeywords)`

  This method adds the same callback into both sides of the processing chain at both points. Keep in mind that the type of the first argument is indeterminate if you use this method! Use it for `finally:` style blocks.

Chaining Deferreds

If you need one Deferred to wait on another, all you need to do is return a Deferred from a method added to addCallbacks. Specifically, if you return Deferred B from a method added to Deferred A using A.addCallbacks, Deferred A's processing chain will stop until Deferred B's .callback() method is called; at that point, the next callback in A will be passed the result of the last callback in Deferred B's processing chain at the time.

Note: If a Deferred is somehow returned from its *own* callbacks (directly or indirectly), the behavior is undefined. The Deferred code will make an attempt to detect this situation and produce a warning. In the future, this will become an exception.

If this seems confusing, don't worry about it right now – when you run into a situation where you need this behavior, you will probably recognize it immediately and realize why this happens. If you want to chain deferreds manually, there is also a convenience method to help you.

- `chainDeferred(otherDeferred)`

  Add `otherDeferred` to the end of this Deferred's processing chain. When self.callback is called, the result of my processing chain up to this point will be passed to `otherDeferred.callback`. Further additions to my callback chain do not affect `otherDeferred`.

  This is the same as `self.addCallbacks(otherDeferred.callback, otherDeferred.errback)`.

See also

1. *Generating Deferreds*, an introduction to writing asynchronous functions that return Deferreds.

Generating Deferreds

Deferred objects are signals that a function you have called does not yet have the data you want available. When a function returns a Deferred object, your calling function attaches callbacks to it to handle the data when available.

This document addresses the other half of the question: writing functions that return Deferreds, that is, constructing Deferred objects, arranging for them to be returned immediately without blocking until data is available, and firing their callbacks when the data is available.

This document assumes that you are familiar with the asynchronous model used by Twisted, and with *using deferreds returned by functions* .

Class overview

This is an overview API reference for Deferred from the point of creating a Deferred and firing its callbacks and errbacks. It is not meant to be a substitute for the docstrings in the Deferred class, but can provide guidelines for its use.

There is a parallel overview of functions used by calling function which the Deferred is returned to at *Using Deferreds* .

Basic Callback Functions

- `callback(result)`

  Run success callbacks with the given result. *This can only be run once.* Later calls to this or `errback` will raise twisted.internet.defer.AlreadyCalledError . If further callbacks or errbacks are added after this point, addCallbacks will run the callbacks immediately.

- `errback(failure)`

  Run error callbacks with the given failure. *This can only be run once.* Later calls to this or `callback` will raise twisted.internet.defer.AlreadyCalledError . If further callbacks or errbacks are added after this point, addCallbacks will run the callbacks immediately.

What Deferreds don't do: make your code asynchronous

Deferreds do not make the code magically not block.

Let's take this function as an example:

```
from twisted.internet import defer

TARGET = 10000

def largeFibonnaciNumber():
    # create a Deferred object to return:
    d = defer.Deferred()

    # calculate the ten thousandth Fibonnaci number

    first = 0
    second = 1

    for i in range(TARGET - 1):
        new = first + second
```

```
        first = second
        second = new
        if i % 100 == 0:
            print("Progress: calculating the %dth Fibonnaci number" % i)

    # give the Deferred the answer to pass to the callbacks:
    d.callback(second)

    # return the Deferred with the answer:
    return d

import time

timeBefore = time.time()

# call the function and get our Deferred
d = largeFibonnaciNumber()

timeAfter = time.time()

print("Total time taken for largeFibonnaciNumber call: %0.3f seconds" % \
      (timeAfter - timeBefore))

# add a callback to it to print the number

def printNumber(number):
    print("The %dth Fibonacci number is %d" % (TARGET, number))

print("Adding the callback now.")

d.addCallback(printNumber)
```

You will notice that despite creating a Deferred in the `largeFibonnaciNumber` function, these things happened:

- the "Total time taken for largeFibonnaciNumber call" output shows that the function did not return immediately as asynchronous functions are expected to do; and
- rather than the callback being added before the result was available and called after the result is available, it isn't even added until after the calculation has been completed.

The function completed its calculation before returning, blocking the process until it had finished, which is exactly what asynchronous functions are not meant to do. Deferreds are not a non-blocking talisman: they are a signal for asynchronous functions to *use* to pass results onto callbacks, but using them does not guarantee that you have an asynchronous function.

Advanced Processing Chain Control

- `pause()`

  Cease calling any methods as they are added, and do not respond to `callback` , until `self.unpause()` is called.

- `unpause()`

  If `callback` has been called on this Deferred already, call all the callbacks that have been added to this Deferred since `pause` was called.

  Whether it was called or not, this will put this Deferred in a state where further calls to `addCallbacks` or `callback` will work as normal.

Returning Deferreds from synchronous functions

Sometimes you might wish to return a Deferred from a synchronous function. There are several reasons why, the major two are maintaining API compatibility with another version of your function which returns a Deferred, or allowing for the possibility that in the future your function might need to be asynchronous.

In the *Using Deferreds* reference, we gave the following example of a synchronous function:

synch-validation.py

```
def synchronousIsValidUser(user):
    '''
    Return true if user is a valid user, false otherwise
    '''
    return user in ["Alice", "Angus", "Agnes"]
```

While we can require that callers of our function wrap our synchronous result in a Deferred using maybeDeferred , for the sake of API compatibility it is better to return a Deferred ourselves using defer.succeed :

```
from twisted.internet import defer

def immediateIsValidUser(user):
    '''
    Returns a Deferred resulting in true if user is a valid user, false
    otherwise
    '''

    result = user in ["Alice", "Angus", "Agnes"]

    # return a Deferred object already called back with the value of result
    return defer.succeed(result)
```

There is an equivalent defer.fail method to return a Deferred with the errback chain already fired.

Integrating blocking code with Twisted

At some point, you are likely to need to call a blocking function: many functions in third party libraries will have long running blocking functions. There is no way to 'force' a function to be asynchronous: it must be written that way specifically. When using Twisted, your own code should be asynchronous, but there is no way to make third party functions asynchronous other than rewriting them.

In this case, Twisted provides the ability to run the blocking code in a separate thread rather than letting it block your application. The twisted.internet.threads.deferToThread function will set up a thread to run your blocking function, return a Deferred and later fire that Deferred when the thread completes.

Let's assume our `largeFibonnaciNumber` function from above is in a third party library (returning the result of the calculation, not a Deferred) and is not easily modifiable to be finished in discrete blocks. This example shows it being called in a thread, unlike in the earlier section we'll see that the operation does not block our entire program:

```
def largeFibonnaciNumber():
    """
    Represent a long running blocking function by calculating
    the TARGETth Fibonnaci number
    """
    TARGET = 10000

    first = 0
    second = 1
```

```
    for i in range(TARGET - 1):
        new = first + second
        first = second
        second = new

    return second

from twisted.internet import threads, reactor

def fibonacciCallback(result):
    """
    Callback which manages the largeFibonnaciNumber result by
    printing it out
    """
    print("largeFibonnaciNumber result =", result)
    # make sure the reactor stops after the callback chain finishes,
    # just so that this example terminates
    reactor.stop()

def run():
    """
    Run a series of operations, deferring the largeFibonnaciNumber
    operation to a thread and performing some other operations after
    adding the callback
    """
    # get our Deferred which will be called with the largeFibonnaciNumber result
    d = threads.deferToThread(largeFibonnaciNumber)
    # add our callback to print it out
    d.addCallback(fibonacciCallback)
    print("1st line after the addition of the callback")
    print("2nd line after the addition of the callback")

if __name__ == '__main__':
    run()
    reactor.run()
```

Possible sources of error

Deferreds greatly simplify the process of writing asynchronous code by providing a standard for registering callbacks, but there are some subtle and sometimes confusing rules that you need to follow if you are going to use them. This mostly applies to people who are writing new systems that use Deferreds internally, and not writers of applications that just add callbacks to Deferreds produced and processed by other systems. Nevertheless, it is good to know.

Firing Deferreds more than once is impossible

Deferreds are one-shot. You can only call `Deferred.callback` or `Deferred.errback` once. The processing chain continues each time you add new callbacks to an already-called-back-to Deferred.

Synchronous callback execution

If a Deferred already has a result available, `addCallback` **may** call the callback synchronously: that is, immediately after it's been added. In situations where callbacks modify state, it is might be desirable for the chain of processing to

halt until all callbacks are added. For this, it is possible to `pause` and `unpause` a Deferred's processing chain while you are adding lots of callbacks.

Be careful when you use these methods! If you `pause` a Deferred, it is *your* responsibility to make sure that you unpause it. The function adding the callbacks must unpause a paused Deferred, it should *never* be the responsibility of the code that actually fires the callback chain by calling `callback` or `errback` as this would negate its usefulness!

Scheduling tasks for the future

Let's say we want to run a task X seconds in the future. The way to do that is defined in the reactor interface twisted.internet.interfaces.IReactorTime :

```
from twisted.internet import reactor

def f(s):
    print("this will run 3.5 seconds after it was scheduled: %s" % s)

reactor.callLater(3.5, f, "hello, world")

# f() will only be called if the event loop is started.
reactor.run()
```

If the result of the function is important or if it may be necessary to handle exceptions it raises, then the twisted.internet.task.deferLater utility conveniently takes care of creating a Deferred and setting up a delayed call:

```
from twisted.internet import task
from twisted.internet import reactor

def f(s):
    return "This will run 3.5 seconds after it was scheduled: %s" % s

d = task.deferLater(reactor, 3.5, f, "hello, world")
def called(result):
    print(result)
d.addCallback(called)

# f() will only be called if the event loop is started.
reactor.run()
```

If we want a task to run every X seconds repeatedly, we can use twisted.internet.task.LoopingCall:

```
from twisted.internet import task
from twisted.internet import reactor

loopTimes = 3
failInTheEnd = False
_loopCounter = 0

def runEverySecond():
    """
    Called at ever loop interval.
    """
    global _loopCounter

    if _loopCounter < loopTimes:
        _loopCounter += 1
        print('A new second has passed.')
```

```
        return

    if failInTheEnd:
        raise Exception('Failure during loop execution.')

    # We looped enough times.
    loop.stop()
    return


def cbLoopDone(result):
    """
    Called when loop was stopped with success.
    """
    print("Loop done.")
    reactor.stop()


def ebLoopFailed(failure):
    """
    Called when loop execution failed.
    """
    print(failure.getBriefTraceback())
    reactor.stop()


loop = task.LoopingCall(runEverySecond)

# Start looping every 1 second.
loopDeferred = loop.start(1.0)

# Add callbacks for stop and failure.
loopDeferred.addCallback(cbLoopDone)
loopDeferred.addErrback(ebLoopFailed)

reactor.run()
```

If we want to cancel a task that we've scheduled:

```
from twisted.internet import reactor

def f():
    print("I'll never run.")

callID = reactor.callLater(5, f)
callID.cancel()
reactor.run()
```

As with all reactor-based code, in order for scheduling to work the reactor must be started using `reactor.run()`.

See also

1. *Timing out Deferreds*

Using Threads in Twisted

How Twisted Uses Threads Itself

All callbacks registered with the reactor - for example, `dataReceived`, `connectionLost`, or any higher-level method that comes from these, such as `render_GET` in twisted.web, or a callback added to a `Deferred` - are called from `reactor.run`. The terminology around this is that we say these callbacks are run in the "main thread", or "reactor thread" or "I/O thread".

Therefore, internally, Twisted makes very little use of threads. This is not to say that it makes *no* use of threads; there are plenty of APIs which have no non-blocking equivalent, so when Twisted needs to call those, it calls them in a thread. One prominent example of this is system hostname resolution: unless you have configured Twisted to use its own DNS client in `twisted.names`, it will have to use your operating system's blocking APIs to map host names to IP addresses, in the reactor's thread pool. However, this is something you only need to know about for resource-tuning purposes, like setting the number of threads to use; otherwise, it is an implementation detail you can ignore.

It is a common mistake is to think that because Twisted can manage multiple connections once, things are happening in multiple threads, and so you need to carefully manage locks. Lucky for you, Twisted does most things in one thread! This document will explain how to interact with existing APIs which need to be run within their own threads because they block. If you're just using Twisted's own APIs, the rule for threads is simply "don't use them".

Invoking Twisted From Other Threads

Methods within Twisted may only be invoked from the reactor thread unless otherwise noted. Very few things within Twisted are thread-safe. For example, writing data to a transport from a protocol is not thread-safe. This means that if you start a thread and call a Twisted method, you might get correct behavior... or you might get hangs, crashes, or corrupted data. So don't do it.

The right way to call methods on the reactor from another thread, and therefore any objects which might call methods on the reactor, is to give a function to the reactor to execute within its own thread. This can be done using the function callFromThread:

```
from twisted.internet import reactor
def notThreadSafe(someProtocol, message):
    someProtocol.transport.write(b"a message: " + message)
def callFromWhateverThreadYouWant():
    reactor.callFromThread(notThreadSafe, b"hello")
```

In this example, `callFromWhateverThreadYouWant` is thread-safe and can be invoked by any thread, but `notThreadSafe` should only ever be called by code running in the thread where `reactor.run` is running.

Note: There are many objects within Twisted that represent values - for example, FilePath and URLPath - which you may construct yourself. These may be safely constructed and used within a non-reactor thread as long as they are not shared with other threads. However, you should be sure that these objects do not share any state, especially not with the reactor. One good rule of thumb is that any object whose methods return `Deferred`s is almost certainly touching the reactor at some point, and should never be accessed from a non-reactor thread.

Running Code In Threads

Sometimes we may want to run code in a non-reactor thread, to avoid blocking the reactor. Twisted provides an API for doing so, the callInThread method on the reactor.

For example, to run a method in a non-reactor thread we can do:

```
from __future__ import print_function
from twisted.internet import reactor

def aSillyBlockingMethod(x):
    import time
    time.sleep(2)
    print(x)

reactor.callInThread(aSillyBlockingMethod, "2 seconds have passed")
reactor.run()
```

`callInThread` will put your code into a queue, to be run by the next available thread in the reactor's thread pool. This means that depending on what other work has been submitted to the pool, your method may not run immediately.

Note: Keep in mind that `callInThread` can only concurrently run a fixed maximum number of tasks, and all users of the reactor are sharing that limit. Therefore, you should not submit *tasks which depend on other tasks in order to complete* to be executed by `callInThread`. An example of such a task would be something like this:

```
from __future__ import print_function

q = Queue()
def blocker():
    print(q.get() + q.get())
def unblocker(a, b):
    q.put(a)
    q.put(b)
```

In this case, `blocker` will block *forever* unless `unblocker` can successfully run to give it inputs; similarly, `unblocker` might block forever if `blocker` is not run to consume its outputs. So if you had a threadpool of maximum size X, and you ran `for each in range(X): reactor.callInThread(blocker)`, the reactor threadpool would be wedged forever, unable to process more work or even shut down.

See "Managing the Reactor Thread Pool" below to tune these limits.

Getting Results

callInThread and callFromThread allow you to move the execution of your code out of and into the reactor thread, respectively, but that isn't always enough.

When we run some code, we often want to know what its result was. For this, Twisted provides two methods: deferToThread and blockingCallFromThread, defined in the `twisted.internet.threads` module.

To get a result from some blocking code back into the reactor thread, we can use deferToThread to execute it instead of callFromThread.

> from __future__ import print_function from twisted.internet import reactor, threads
>
> **def doLongCalculation():** # .... do long calculation here ... return 3
>
> **def printResult(x):** print(x)
>
> # run method in thread and get result as defer.Deferred d = threads.deferToThread(doLongCalculation) d.addCallback(printResult) reactor.run()

Similarly, you want some code running in a non-reactor thread wants to invoke some code in the reactor thread and get its result, you can use blockingCallFromThread:

```
from twisted.internet import threads, reactor, defer
from twisted.web.client import Agent
from twisted.web.error import Error

def inThread():
    agent = Agent(reactor)
    try:
        result = threads.blockingCallFromThread(
            reactor, agent.request, "GET", "http://twistedmatrix.com/"))
    except Error as exc:
        print(exc)
    else:
        print(result)
    reactor.callFromThread(reactor.stop)

reactor.callInThread(inThread)
reactor.run()
```

`blockingCallFromThread` will return the object or raise the exception returned or raised by the function passed to it. If the function passed to it returns a Deferred, it will return the value the Deferred is called back with or raise the exception it is errbacked with.

Managing the Reactor Thread Pool

We may want to modify the size of the thread pool, increasing or decreasing the number of threads in use. We can do this:

```
from twisted.internet import reactor

reactor.suggestThreadPoolSize(30)
```

The default size of the thread pool depends on the reactor being used; the default reactor uses a minimum size of 0 and a maximum size of 10.

The reactor thread pool is implemented by ThreadPool. To access methods on this object for more advanced tuning and monitoring (see the API documentation for details) you can get the thread pool with getThreadPool.

Producers and Consumers: Efficient High-Volume Streaming

The purpose of this guide is to describe the Twisted *producer* and *consumer* system. The producer system allows applications to stream large amounts of data in a manner which is both memory and CPU efficient, and which does not introduce a source of unacceptable latency into the reactor.

Readers should have at least a passing familiarity with the terminology associated with interfaces.

Push Producers

A push producer is one which will continue to generate data without external prompting until told to stop; a pull producer will generate one chunk of data at a time in response to an explicit request for more data.

The push producer API is defined by the IPushProducer interface. It is best to create a push producer when data generation is closedly tied to an event source. For example, a proxy which forwards incoming bytes from one socket to another outgoing socket might be implemented using a push producer: the dataReceived takes the role of an event source from which the producer generates bytes, and requires no external intervention in order to do so.

There are three methods which may be invoked on a push producer at various points in its lifetime: pauseProducing , resumeProducing , and stopProducing .

pauseProducing()

In order to avoid the possibility of using an unbounded amount of memory to buffer produced data which cannot be processed quickly enough, it is necessary to be able to tell a push producer to stop producing data for a while. This is done using the `pauseProducing` method. Implementers of a push producer should temporarily stop producing data when this method is invoked.

resumeProducing()

After a push producer has been paused for some time, the excess of data which it produced will have been processed and the producer may again begin producing data. When the time for this comes, the push producer will have `resumeProducing` invoked on it.

stopProducing()

Most producers will generate some finite (albeit, perhaps, unknown in advance) amount of data and then stop, having served their intended purpose. However, it is possible that before this happens an event will occur which renders the remaining, unproduced data irrelevant. In these cases, producing it anyway would be wasteful. The `stopProducing` method will be invoked on the push producer. The implementation should stop producing data and clean up any resources owned by the producer.

Pull Producers

The pull producer API is defined by the IPullProducer interface. Pull producers are useful in cases where there is no clear event source involved with the generation of data. For example, if the data is the result of some algorithmic process that is bound only by CPU time, a pull producer is appropriate.

Pull producers are defined in terms of only two methods: resumeProducing and stopProducing .

resumeProducing()

Unlike push producers, a pull producer is expected to **only** produce data in response to `resumeProducing` being called. This method will be called whenever more data is required. How much data to produce in response to this method call depends on various factors: too little data and runtime costs will be dominated by the back-and-forth event notification associated with a buffer becoming empty and requesting more data to process; too much data and memory usage will be driven higher than it needs to be and the latency associated with creating so much data will cause overall performance in the application to suffer. A good rule of thumb is to generate between 16 and 64 kilobytes of data at a time, but you should experiment with various values to determine what is best for your application.

stopProducing()

This method has the same meaning for pull producers as it does for push producers.

Consumers

This far, I've discussed the various external APIs of the two kinds of producers supported by Twisted. However, I have not mentioned where the data a producer generates actually goes, nor what entity is responsible for invoking these APIs. Both of these roles are filled by *consumers* . Consumers are defined by the one interface IConsumer .

`IConsumer` , defines three methods: registerProducer , unregisterProducer , and write .

registerProducer(producer, streaming)

So that a consumer can invoke methods on a producer, the consumer needs to be told about the producer. This is done with the `registerProducer` method. The first argument is either a `IPullProducer` or `IPushProducer` provider; the second argument indicates which of these interfaces is provided: `True` for push producers, `False` for pull producers.

unregisterProducer()

Eventually a consumer will not longer be interested in a producer. This could be because the producer has finished generating all its data, or because the consumer is moving on to something else, or any number of other reasons. In any case, this method reverses the effects of `registerProducer` .

write(data)

As you might guess, this is the method which a producer calls when it has generated some data. Push producers should call it as frequently as they like as long as they are not paused. Pull producers should call it once for each time `resumeProducing` is called on them.

Further Reading

An example push producer application can be found in `doc/examples/streaming.py` .

- *Components: Interfaces and Adapters*
- FileSender : A Simple Pull Producer

Choosing a Reactor and GUI Toolkit Integration

Overview

Twisted provides a variety of implementations of the twisted.internet.reactor . The specialized implementations are suited for different purposes and are designed to integrate better with particular platforms.

The *epoll()-based reactor* is Twisted's default on Linux. Other platforms use *poll()* , or the most cross-platform reactor, *select()* .

Platform-specific reactor implementations exist for:

- *Poll for Linux*
- *Epoll for Linux 2.6*
- *WaitForMultipleObjects (WFMO) for Win32*
- *Input/Output Completion Port (IOCP) for Win32*

- *KQueue for FreeBSD and Mac OS X*
- *CoreFoundation for Mac OS X*

The remaining custom reactor implementations provide support for integrating with the native event loops of various graphical toolkits. This lets your Twisted application use all of the usual Twisted APIs while still being a graphical application.

Twisted currently integrates with the following graphical toolkits:

- *GTK+ 2.0*
- *GTK+ 3.0 and GObject Introspection*
- *Tkinter*
- *wxPython*
- *Win32*
- *CoreFoundation*
- *PyUI*

When using applications that are runnable using `twistd`, e.g. TACs or plugins, there is no need to choose a reactor explicitly, since this can be chosen using `twistd` 's -r option.

In all cases, the event loop is started by calling `reactor.run()`. In all cases, the event loop should be stopped with `reactor.stop()`.

IMPORTANT: installing a reactor should be the first thing done in the app, since any code that does `from twisted.internet import reactor` will automatically install the default reactor if the code hasn't already installed one.

Reactor Functionality

	Status	TCP	SSL	UDP	Threading	Processes	Scheduling	Platforms
select()	Stable	Y	Y	Y	Y	Y	Y	Unix, Win32
poll	Stable	Y	Y	Y	Y	Y	Y	Unix
WaitForMultipleObjects (WFMO) for Win32	Experimental	Y	Y	Y	Y	Y	Y	Win32
Input/Output Completion Port (IOCP) for Win32	Experimental	Y	Y	Y	Y	Y	Y	Win32
CoreFoundation	Unmaintained	Y	Y	Y	Y	Y	Y	Mac OS X
epoll	Stable	Y	Y	Y	Y	Y	Y	Linux 2.6
GTK+	Stable	Y	Y	Y	Y	Y	Y	Unix, Win32
wx	Experimental	Y	Y	Y	Y	Y	Y	Unix, Win32
kqueue	Stable	Y	Y	Y	Y	Y	Y	FreeBSD

General Purpose Reactors

Select()-based Reactor

The `select` reactor is the default on platforms that don't provide a better alternative that covers all use cases. If the `select` reactor is desired, it may be installed via:

```
from twisted.internet import selectreactor
selectreactor.install()

from twisted.internet import reactor
```

Platform-Specific Reactors

Poll-based Reactor

The PollReactor will work on any platform that provides `select.poll` . With larger numbers of connected sockets, it may provide for better performance than the SelectReactor.

```
from twisted.internet import pollreactor
pollreactor.install()

from twisted.internet import reactor
```

KQueue

The KQueue Reactor allows Twisted to use FreeBSD's kqueue mechanism for event scheduling. See instructions in the twisted.internet.kqreactor 's docstring for installation notes.

```
from twisted.internet import kqreactor
kqreactor.install()

from twisted.internet import reactor
```

WaitForMultipleObjects (WFMO) for Win32

The Win32 reactor is not yet complete and has various limitations and issues that need to be addressed. The reactor supports GUI integration with the win32gui module, so it can be used for native Win32 GUI applications.

```
from twisted.internet import win32eventreactor
win32eventreactor.install()

from twisted.internet import reactor
```

Input/Output Completion Port (IOCP) for Win32

Windows provides a fast, scalable event notification system known as IO Completion Ports, or IOCP for short. Twisted includes a reactor based on IOCP which is nearly complete.

```
from twisted.internet import iocpreactor
iocpreactor.install()
```

```
from twisted.internet import reactor
```

Epoll-based Reactor

The EPollReactor will work on any platform that provides `epoll` , today only Linux 2.6 and over. The implementation of the epoll reactor currently uses the Level Triggered interface, which is basically like poll() but scales much better.

```
from twisted.internet import epollreactor
epollreactor.install()

from twisted.internet import reactor
```

GUI Integration Reactors

GTK+

Twisted integrates with PyGTK version 2.0 using the `gtk2reactor` . An example Twisted application that uses GTK+ can be found in `doc/core/examples/pbgtk2.py` .

GTK-2.0 split the event loop out of the GUI toolkit and into a separate module called "glib" . To run an application using the glib event loop, use the `glib2reactor` . This will be slightly faster than `gtk2reactor` (and does not require a working X display), but cannot be used to run GUI applications.

```
from twisted.internet import gtk2reactor # for gtk-2.0
gtk2reactor.install()

from twisted.internet import reactor
```

```
from twisted.internet import glib2reactor # for non-GUI apps
glib2reactor.install()

from twisted.internet import reactor
```

GTK+ 3.0 and GObject Introspection

Twisted integrates with GTK+ 3 and GObject through PyGObject's introspection using the `gtk3reactor` and `gireactor` reactors.

```
from twisted.internet import gtk3reactor
gtk3reactor.install()

from twisted.internet import reactor
```

```
from twisted.internet import gireactor # for non-GUI apps
gireactor.install()

from twisted.internet import reactor
```

GLib 3.0 introduces the concept of `GApplication` , a class that handles application uniqueness in a cross-platform way and provides its own main loop. Its counterpart `GtkApplication` also handles application lifetime with

respect to open windows. Twisted supports registering these objects with the event loop, which should be done before running the reactor:

```
from twisted.internet import gtk3reactor
gtk3reactor.install()

from gi.repository import Gtk
app = Gtk.Application(...)

from twisted import reactor
reactor.registerGApplication(app)
reactor.run()
```

wxPython

Twisted currently supports two methods of integrating wxPython. Unfortunately, neither method will work on all wxPython platforms (such as GTK2 or Windows). It seems that the only portable way to integrate with wxPython is to run it in a separate thread. One of these methods may be sufficient if your wx app is limited to a single platform.

As with *Tkinter* , the support for integrating Twisted with a wxPython application uses specialized support code rather than a simple reactor.

```
from wxPython.wx import *
from twisted.internet import wxsupport, reactor

myWxAppInstance = wxApp(0)
wxsupport.install(myWxAppInstance)
```

However, this has issues when running on Windows, so Twisted now comes with alternative wxPython support using a reactor. Using this method is probably better. Initialization is done in two stages. In the first, the reactor is installed:

```
from twisted.internet import wxreactor
wxreactor.install()

from twisted.internet import reactor
```

Later, once a `wxApp` instance has been created, but before `reactor.run()` is called:

```
from twisted.internet import reactor
myWxAppInstance = wxApp(0)
reactor.registerWxApp(myWxAppInstance)
```

An example Twisted application that uses wxPython can be found in `doc/core/examples/wxdemo.py` .

CoreFoundation

Twisted integrates with PyObjC version 1.0. Sample applications using Cocoa and Twisted are available in the examples directory under `doc/core/examples/threadedselect/Cocoa` .

```
from twisted.internet import cfreactor
cfreactor.install()

from twisted.internet import reactor
```

Non-Reactor GUI Integration

Tkinter

The support for Tkinter doesn't use a specialized reactor. Instead, there is some specialized support code:

```
from Tkinter import *
from twisted.internet import tksupport, reactor

root = Tk()

# Install the Reactor support
tksupport.install(root)

# at this point build Tk app as usual using the root object,
# and start the program with "reactor.run()", and stop it
# with "reactor.stop()".
```

PyUI

As with *Tkinter* , the support for integrating Twisted with a PyUI application uses specialized support code rather than a simple reactor.

```
from twisted.internet import pyuisupport, reactor

pyuisupport.install(args=(640, 480), kw={'renderer': 'gl'})
```

An example Twisted application that uses PyUI can be found in `doc/core/examples/pyuidemo.py` .

Getting Connected with Endpoints

Introduction

On a network, one can think of any given connection as a long wire, stretched between two points. Lots of stuff can happen along the length of that wire - routers, switches, network address translation, and so on, but that is usually invisible to the application passing data across it. Twisted strives to make the nature of the "wire" as transparent as possible, with highly abstract interfaces for passing and receiving data, such as ITransport and IProtocol.

However, the application can't be completely ignorant of the wire. In particular, it must do something to *start* the connection, and to do so, it must identify the *end points* of the wire. There are different names for the roles of each end point - "initiator" and "responder", "connector" and "listener", or "client" and "server" - but the common theme is that one side of the connection waits around for someone to connect to it, and the other side does the connecting.

In Twisted 10.1, several new interfaces were introduced to describe each of these roles for stream-oriented connections: IStreamServerEndpoint and IStreamClientEndpoint. The word "stream", in this case, refers to endpoints which treat a connection as a continuous stream of bytes, rather than a sequence of discrete datagrams: TCP is a "stream" protocol whereas UDP is a "datagram" protocol.

Constructing and Using Endpoints

In both *Writing Servers* and *Writing Clients*, we covered basic usage of endpoints; you construct an appropriate type of server or client endpoint, and then call `listen` (for servers) or `connect` (for clients).

In both of those tutorials, we constructed specific types of endpoints directly. However, in most programs, you will want to allow the user to specify where to listen or connect, in a way which will allow the user to request different strategies, without having to adjust your program. In order to allow this, you should use clientFromString or serverFromString.

There's Not Much To It

Each type of endpoint is just an interface with a single method that takes an argument. `serverEndpoint.listen(factory)` will start listening on that endpoint with your protocol factory, and `clientEndpoint.connect(factory)` will start a single connection attempt. Each of these APIs returns a value, though, which can be important.

However, if you are not already, you *should* be very familiar with *Deferreds*, as they are returned by both `connect` and `listen` methods, to indicate when the connection has connected or the listening port is up and running.

Servers and Stopping

IStreamServerEndpoint.listen returns a Deferred that fires with an IListeningPort. Note that this deferred may errback. The most common cause of such an error would be that another program is already using the requested port number, but the exact cause may vary depending on what type of endpoint you are listening on. If you receive such an error, it means that your application is not actually listening, and will not receive any incoming connections. It's important to somehow alert an administrator of your server, in this case, especially if you only have one listening port!

Note also that once this has succeeded, it will continue listening forever. If you need to *stop* listening for some reason, in response to anything other than a full server shutdown (`reactor.stop` and / or `twistd` will usually handle that case for you), make sure you keep a reference around to that listening port object so you can call IListeningPort.stopListening on it. Finally, keep in mind that `stopListening` itself returns a `Deferred`, and the port may not have fully stopped listening until that `Deferred` has fired.

Most server applications will not need to worry about these details. One example of a case where you would need to be concerned with all of these events would be an implementation of a protocol like non-`PASV` FTP, where new listening ports need to be bound for the lifetime of a particular action, then disposed of.

Clients and Cancelling

connectProtocol connects a Protocol instance to a given IStreamClientEndpoint. It returns a `Deferred` which fires with the `Protocol` once the connection has been made. Connection attempts may fail, and so that Deferred may also errback. If it does so, you will have to try again; no further attempts will be made. See the *client documentation* for an example use.

connectProtocol is a wrapper around a lower-level API: IStreamClientEndpoint.connect will use a protocol factory for a new outgoing connection attempt. It returns a `Deferred` which fires with the `IProtocol` returned from the factory's `buildProtocol` method, or errbacks with the connection failure.

Connection attempts may also take a long time, and your users may become bored and wander off. If this happens, and your code decides, for whatever reason, that you've been waiting for the connection too long, you can call Deferred.cancel on the `Deferred` returned from connect or connectProtocol, and the underlying machinery should give up on the connection. This should cause the `Deferred` to errback, usually with CancelledError; although you should consult the documentation for your particular endpoint type to see if it may do something different.

Although some endpoint types may imply a built-in timeout, the interface does not guarantee one. If you don't have any way for the application to cancel a wayward connection attempt, the attempt may just keep waiting forever. For example, a very simple 30-second timeout could be implemented like this:

```
attempt = connectProtocol(myEndpoint, myProtocol)
reactor.callLater(30, attempt.cancel)
```

Note: If you've used `ClientFactory` before, keep in mind that the `connect` method takes a `Factory`, not a `ClientFactory`. Even if you pass a `ClientFactory` to `endpoint.connect`, its `clientConnectionFailed` and `clientConnectionLost` methods will not be called. In particular, clients that extend `ReconnectingClientFactory` won't reconnect. The next section describes how to set up reconnecting clients on endpoints.

Persistent Client Connections

twisted.application.internet.ClientService can maintain a persistent outgoing connection to a server which can be started and stopped along with your application.

One popular protocol to maintain a long-lived client connection to is IRC, so for an example of `ClientService`, here's how you would make a long-lived encrypted connection to an IRC server (other details, like how to authenticate, omitted for brevity):

```
from twisted.internet.protocol import Factory
from twisted.internet.endpoints import clientFromString
from twisted.words.protocols.irc import IRCClient
from twisted.application.internet import ClientService
from twisted.internet import reactor

myEndpoint = clientFromString(reactor, "tls:example.com:6997")
myFactory = Factory.forProtocol(IRCClient)

myReconnectingService = ClientService(myEndpoint, myFactory)
```

If you already have a parent service, you can add the reconnecting service as a child service:

```
parentService.addService(myReconnectingService)
```

If you do not have a parent service, you can start and stop the reconnecting service using its `startService` and `stopService` methods.

`ClientService.stopService` returns a `Deferred` that fires once the current connection closes or the current connection attempt is cancelled.

Getting The Active Client

When maintaining a long-lived connection, it's often useful to be able to get the current connection (if the connection is active) or wait for the next connection (if a connection attempt is currently in progress). For example, we might want to pass our `ClientService` from the previous example to some code that can send IRC notifications in response to some external event. The `ClientService.whenConnected` method returns a `Deferred` that fires with the next available `Protocol` instance. You can use it like so:

```
waitForConnection = myReconnectingService.whenConnected()
def connectedNow(clientForIRC):
    clientForIRC.say("#bot-test", "hello, world!")
waitForConnection.addCallback(connectedNow)
```

Keep in mind that you may need to wrap this up for your particular application, since when no existing connection is available, the callback is executed just as soon as the connection is established. For example, that little snippet is slightly oversimplified: at the time `connectedNow` is run, the bot hasn't authenticated or joined the channel yet, so its message will be refused. A real-life IRC bot would need to have its own method for waiting until the connection is fully ready for chat before chatting.

Reporting an Initial Failure

Often times, a failure of the very first connection attempt is special. It may indicate a problem that won't go away by just trying harder. The service may be configured with the wrong hostname, or the user may not have an internet connection at all (perhaps they forgot to turn on their wifi adapter).

Applications can ask `whenConnected` to make their `Deferred` fail if the service makes one or more connection attempts in a row without success. You can pass the `failAfterFailures` parameter into `ClientService` to set this threshold.

By calling `whenConnected(failAfterFailures=1)` when the service is first started (just before or just after `startService`), your application will get notification of an initial connection failure.

Setting it to 1 makes it fail after a single connection failure. Setting it to 2 means it will try once, wait a bit, try again, and then either fail or succeed depending upon the outcome of the second connection attempt. You can use 3 or more too, if you're feeling particularly patient. The default of `None` means it will wait forever for a successful connection.

Regardless of `failAfterFailures`, the `Deferred` will always fail with CancelledError if the service is stopped before a connection is made.

```
waitForConnection = myReconnectingService.whenConnected(failAfterFailures=1)
def connectedNow(clientForIRC):
    clientForIRC.say("#bot-test", "hello, world!")
def failed(f):
    print("initial connection failed: %s" % (f,))
    # now you should stop the service and report the error upwards
waitForConnection.addCallbacks(connectedNow, failed)
```

Retry Policies

`ClientService` will immediately attempt an outgoing connection when `startService` is called. If that connection attempt fails for any reason (name resolution, connection refused, network unreachable, and so on), it will retry according to the policy specified in the `retryPolicy` constructor argument. By default, `ClientService` will use an exponential backoff algorithm with a minimum delay of 1 second and a maximum delay of 1 minute, and a jitter of up to 1 additional second to prevent stampeding-herd performance cascades. This is a good default, and if you do not have highly specialized requirements, you probably want to use it. If you need to tune these parameters, you have two options:

1. You can pass your own timeout policy to `ClientService`'s constructor. A timeout policy is a callable that takes the number of failed attempts, and computes a delay until the next connection attempt. So, for example, if you are *really really sure* that you want to reconnect *every single second* if the service you are talking to goes down, you can do this:

   ```
   myReconnectingService = ClientService(myEndpoint, myFactory, retryPolicy=lambda␣
   ↪ignored: 1)
   ```

   Of course, unless you have only one client and only one server and they're both on localhost, this sort of policy is likely to cause massive performance degradation and thundering herd resource contention in the event of your server's failure, so you probably want to take the second option...

2. You can tweak the default exponential backoff policy with a few parameters by passing the result of twisted.application.internet.backoffPolicy to the `retryPolicy` argument. For example, if you want to make it triple the delay between attempts, but start with a faster connection interval (half a second instead of one second), you could do it like so:

```
myReconnectingService = ClientService(
    myEndpoint, myFactory,
    retryPolicy=backoffPolicy(initialDelay=0.5, factor=3.0)
)
```

Note: Before endpoints, reconnecting clients were created as subclasses of `ReconnectingClientFactory`. These subclasses were required to call `resetDelay`. One of the many advantages of using endpoints is that these special subclasses are no longer needed. `ClientService` accepts ordinary `IProtocolFactory` providers.

Maximizing the Return on your Endpoint Investment

Directly constructing an endpoint in your application is rarely the best option, because it ties your application to a particular type of transport. The strength of the endpoints API is in separating the construction of the endpoint (figuring out where to connect or listen) and its activation (actually connecting or listening).

If you are implementing a library that needs to listen for connections or make outgoing connections, when possible, you should write your code to accept client and server endpoints as parameters to functions or to your objects' constructors. That way, application code that calls your library can provide whatever endpoints are appropriate.

If you are writing an application and you need to construct endpoints yourself, you can allow users to specify arbitrary endpoints described by a string using the clientFromString and serverFromString APIs. Since these APIs just take a string, they provide flexibility: if Twisted adds support for new types of endpoints (for example, IPv6 endpoints, or WebSocket endpoints), your application will automatically be able to take advantage of them with no changes to its code.

Endpoints Aren't Always the Answer

For many use-cases, especially the common case of a `twistd` plugin which runs a long-running server that just binds a simple port, you might not want to use the endpoints APIs directly. Instead, you may want to construct an IService, using strports.service, which will fit neatly into the required structure of *the twistd plugin API*. This doesn't give your application much control - the port starts listening at startup and stops listening at shutdown - but it does provide the same flexibility in terms of what type of server endpoint your application will support.

It is, however, almost always preferable to use an endpoint rather than calling a lower-level APIs like connectTCP, listenTCP, etc, directly. By accepting an arbitrary endpoint rather than requiring a specific reactor interface, you leave your application open to lots of interesting transport-layer extensibility for the future.

Endpoint Types Included With Twisted

The parser used by `clientFromString` and `serverFromString` is extensible via third-party plugins, so the endpoints available on your system depend on what packages you have installed. However, Twisted itself includes a set of basic endpoints that will always be available.

Clients

TCP Supported arguments: `host`, `port`, `timeout`. `timeout` is optional.

For example, `tcp:host=twistedmatrix.com:port=80:timeout=15`.

TLS Required arguments: `host`, `port`.

Optional arguments: `timeout`, `bindAddress`, `certificate`, `privateKey`, `trustRoots`, `endpoint`.

- `host` is a (UTF-8 encoded) hostname to connect to, as well as the host name to verify against.
- `port` is a numeric port number to connect to.
- `timeout` and `bindAddress` have the same meaning as the `timeout` and `bindAddress` for TCP clients.
- `certificate` is the certificate to use for the client; it should be the path name of a PEM file containing a certificate for which `privateKey` is the private key.
- `privateKey` is the client's private key, matching the certificate specified by `certificate`. It should be the path name of a PEM file containing an X.509 client certificate. If `certificate` is specified but `privateKey` is unspecified, Twisted will look for the certificate in the same file as specified by `certificate`.
- `trustRoots` specifies a path to a directory of PEM-encoded certificate files. If you leave this unspecified, Twisted will do its best to use the platform default set of trust roots, which should be the default WebTrust set.
- the optional `endpoint` parameter changes the meaning of the `tls:` endpoint slightly. Rather than the default of connecting over TCP with the same hostname used for verification, you can connect over *any* endpoint type. If you specify the endpoint here, `host` and `port` are used for certificate verification purposes only. Bear in mind you will need to backslash-escape the colons in the endpoint description here.

This client connects to the supplied hostname, validates the server's hostname against the supplied hostname, and then upgrades to TLS immediately after validation succeeds.

The simplest example of this would be: `tls:example.com:443`.

You can use the `endpoint:` feature with TCP if you want to connect to a host name; for example, if your DNS is not working, but you know that the IP address 7.6.5.4 points to `awesome.site.example.com`, you could specify: `tls:awesome.site.example.com:443:endpoint=tcp\:7.6.5.4\:443`.

You can use it with any other endpoint type as well, though; for example, if you had a local UNIX socket that established a tunnel to `awesome.site.example.com` in `/var/run/awesome.sock`, you could instead do `tls:awesome.site.example.com:443:endpoint=unix\:/var/run/awesome.sock`.

Or, from python code:

```
wrapped = HostnameEndpoint('example.com', 443)
contextFactory = optionsForClientTLS(hostname=u'example.com')
endpoint = wrapClientTLS(contextFactory, wrapped)
conn = endpoint.connect(Factory.forProtocol(Protocol))
```

UNIX Supported arguments: `path`, `timeout`, `checkPID`. `path` gives a filesystem path to a listening UNIX domain socket server. `checkPID` (optional) enables a check of the lock file Twisted-based UNIX domain socket servers use to prove they are still running.

For example, `unix:path=/var/run/web.sock`.

TCP (Hostname) Supported arguments: `host`, `port`, `timeout`. `host` is a hostname to connect to. `timeout` is optional. It is a name-based TCP endpoint that returns the connection which is established first amongst the resolved addresses.

For example,

```
endpoint = HostnameEndpoint(reactor, "twistedmatrix.com", 80)
conn = endpoint.connect(Factory.forProtocol(Protocol))
```

SSL (Deprecated)

Note: You should generally prefer the "TLS" client endpoint, above, unless you need to work with versions of Twisted older than 16.0. Among other things:

- the `ssl:` client endpoint requires that you pass ''both" `hostname=` (for hostname verification) as well as `host=` (for a TCP connection address) in order to get hostname verification, which is required for security, whereas `tls:` does the correct thing by default by using the same hostname for both.
- the `ssl:` client endpoint doesn't work with IPv6, and the `tls:` endpoint does.

All TCP arguments are supported, plus: `certKey`, `privateKey`, `caCertsDir`. `certKey` (optional) gives a filesystem path to a certificate (PEM format). `privateKey` (optional) gives a filesystem path to a private key (PEM format). `caCertsDir` (optional) gives a filesystem path to a directory containing trusted CA certificates to use to verify the server certificate.

For example, `ssl:host=twistedmatrix.com:port=443:caCertsDir=/etc/ssl/certs`.

Servers

TCP (IPv4) Supported arguments: `port`, `interface`, `backlog`. `interface` and `backlog` are optional. `interface` is an IP address (belonging to the IPv4 address family) to bind to.

For example, `tcp:port=80:interface=192.168.1.1`.

TCP (IPv6) All TCP (IPv4) arguments are supported, with `interface` taking an IPv6 address literal instead.

For example, `tcp6:port=80:interface=2001\:0DB8\:f00e\:eb00\:\:1`.

SSL All TCP arguments are supported, plus: `certKey`, `privateKey`, `extraCertChain`, `sslmethod`, and `dhParameters`. `certKey` (optional, defaults to the value of privateKey) gives a filesystem path to a certificate (PEM format). `privateKey` gives a filesystem path to a private key (PEM format). `extraCertChain` gives a filesystem path to a file with one or more concatenated certificates in PEM format that establish the chain from a root CA to the one that signed your certificate. `sslmethod` indicates which SSL/TLS version to use (a value like `TLSv1_METHOD`). `dhParameters` gives a filesystem path to a file in PEM format with parameters that are required for Diffie-Hellman key exchange. Since the this is required for the `DHE`-family of ciphers that offer perfect forward secrecy (PFS), it is recommended to specify one. Such a file can be created using `openssl dhparam -out dh_param_1024.pem -2 1024`. Please refer to OpenSSL's documentation on dhparam for further details.

For example, `ssl:port=443:privateKey=/etc/ssl/server.pem:extraCertChain=/etc/ssl/chain.pem:sslmethod=SSLv3_METHOD:dhParameters=dh_param_1024.pem`.

UNIX Supported arguments: `address`, `mode`, `backlog`, `lockfile`. `address` gives a filesystem path to listen on with a UNIX domain socket server. `mode` (optional) gives the filesystem permission/mode (in octal) to apply to that socket. `lockfile` enables use of a separate lock file to prove the server is still running.

For example, `unix:address=/var/run/web.sock:lockfile=1`.

systemd Supported arguments: `domain`, `index`. `domain` indicates which socket domain the inherited file descriptor belongs to (eg INET, INET6). `index` indicates an offset into the array of file descriptors which have been inherited from systemd.

For example, `systemd:domain=INET6:index=3`.

See also *Deploying Twisted with systemd*.

PROXY The PROXY protocol is a stream wrapper and can be applied any of the other server endpoints by placing `haproxy:` in front of a normal port definition.

For example, `haproxy:tcp:port=80:interface=192.168.1.1` or `haproxy:ssl:port=443:privateKey=/etc/ssl/server.pem:extraCertChain=/etc/ssl/chain.pem:sslmethod=SSLv3_METHOD:dhParameters=dh_param_1024.pem`.

The PROXY protocol provides a way for load balancers and reverse proxies to send down the real IP of a connection's source and destination without relying on X-Forwarded-For headers. A Twisted service using this endpoint wrapper must run behind a service that sends valid PROXY protocol headers. For more on the protocol see the formal specification. Both version one and two of the protocol are currently supported.

Components: Interfaces and Adapters

Object oriented programming languages allow programmers to reuse portions of existing code by creating new "classes" of objects which subclass another class. When a class subclasses another, it is said to *inherit* all of its behaviour. The subclass can then "override" and "extend" the behavior provided to it by the superclass. Inheritance is very useful in many situations, but because it is so convenient to use, often becomes abused in large software systems, especially when multiple inheritance is involved. One solution is to use *delegation* instead of "inheritance" where appropriate. Delegation is simply the act of asking *another* object to perform a task for an object. To support this design pattern, which is often referred to as the *components* pattern because it involves many small interacting components, *interfaces* and *adapters* were created by the Zope 3 team.

"Interfaces" are simply markers which objects can use to say "I implement this interface". Other objects may then make requests like "Please give me an object which implements interface X for object type Y". Objects which implement an interface for another object type are called "adapters".

The superclass-subclass relationship is said to be an *is-a* relationship. When designing object hierarchies, object modellers use subclassing when they can say that the subclass *is* the same class as the superclass. For example:

```
class Shape:
    sideLength = 0
    def getSideLength(self):
        return self.sideLength

    def setSideLength(self, sideLength):
        self.sideLength = sideLength

    def area(self):
        raise NotImplementedError("Subclasses must implement area")

class Triangle(Shape):
    def area(self):
        return (self.sideLength * self.sideLength) / 2

class Square(Shape):
    def area(self):
        return self.sideLength * self.sideLength
```

In the above example, a Triangle *is-a* Shape, so it subclasses Shape, and a Square *is-a* Shape, so it also subclasses Shape.

However, subclassing can get complicated, especially when Multiple Inheritance enters the picture. Multiple Inheritance allows a class to inherit from more than one base class. Software which relies heavily on inheritance often ends

up having both very wide and very deep inheritance trees, meaning that one class inherits from many superclasses spread throughout the system. Since subclassing with Multiple Inheritance means *implementation inheritance*, locating a method's actual implementation and ensuring the correct method is actually being invoked becomes a challenge. For example:

```
class Area:
    sideLength = 0
    def getSideLength(self):
        return self.sideLength

    def setSideLength(self, sideLength):
        self.sideLength = sideLength

    def area(self):
        raise NotImplementedError("Subclasses must implement area")

class Color:
    color = None
    def setColor(self, color):
      self.color = color

    def getColor(self):
      return self.color

class Square(Area, Color):
    def area(self):
        return self.sideLength * self.sideLength
```

The reason programmers like using implementation inheritance is because it makes code easier to read since the implementation details of Area are in a separate place than the implementation details of Color. This is nice, because conceivably an object could have a color but not an area, or an area but not a color. The problem, though, is that Square is not really an Area or a Color, but has an area and color. Thus, we should really be using another object oriented technique called *composition*, which relies on delegation rather than inheritance to break code into small reusable chunks. Let us continue with the Multiple Inheritance example, though, because it is often used in practice.

What if both the Color and the Area base class defined the same method, perhaps `calculate`? Where would the implementation come from? The implementation that is located for `Square().calculate()` depends on the method resolution order, or MRO, and can change when programmers change seemingly unrelated things by refactoring classes in other parts of the system, causing obscure bugs. Our first thought might be to change the calculate method name to avoid name clashes, to perhaps `calculateArea` and `calculateColor`. While explicit, this change could potentially require a large number of changes throughout a system, and is error-prone, especially when attempting to integrate two systems which you didn't write.

Let's imagine another example. We have an electric appliance, say a hair dryer. The hair dryer is American voltage. We have two electric sockets, one of them an American 120 Volt socket, and one of them a United Kingdom 240 Volt socket. If we plug the hair dryer into the 240 Volt socket, it is going to expect 120 Volt current and errors will result. Going back and changing the hair dryer to support both `plug120Volt` and `plug240Volt` methods would be tedious, and what if we decided we needed to plug the hair dryer into yet another type of socket? For example:

```
class HairDryer:
    def plug(self, socket):
        if socket.voltage() == 120:
            print("I was plugged in properly and am operating.")
        else:
            print("I was plugged in improperly and ")
            print("now you have no hair dryer any more.")

class AmericanSocket:
```

```
    def voltage(self):
        return 120

class UKSocket:
    def voltage(self):
        return 240
```

Given these classes, the following operations can be performed:

```
>>> hd = HairDryer()
>>> am = AmericanSocket()
>>> hd.plug(am)
I was plugged in properly and am operating.
>>> uk = UKSocket()
>>> hd.plug(uk)
I was plugged in improperly and
now you have no hair dryer any more.
```

We are going to attempt to solve this problem by writing an Adapter for the `UKSocket` which converts the voltage for use with an American hair dryer. An Adapter is a class which is constructed with one and only one argument, the "adaptee" or "original" object. In this example, we will show all code involved for clarity:

```
class AdaptToAmericanSocket:
    def __init__(self, original):
        self.original = original

    def voltage(self):
        return self.original.voltage() / 2
```

Now, we can use it as so:

```
>>> hd = HairDryer()
>>> uk = UKSocket()
>>> adapted = AdaptToAmericanSocket(uk)
>>> hd.plug(adapted)
I was plugged in properly and am operating.
```

So, as you can see, an adapter can 'override' the original implementation. It can also 'extend' the interface of the original object by providing methods the original object did not have. Note that an Adapter must explicitly delegate any method calls it does not wish to modify to the original, otherwise the Adapter cannot be used in places where the original is expected. Usually this is not a problem, as an Adapter is created to conform an object to a particular interface and then discarded.

Interfaces and Components in Twisted code

Adapters are a useful way of using multiple classes to factor code into discrete chunks. However, they are not very interesting without some more infrastructure. If each piece of code which wished to use an adapted object had to explicitly construct the adapter itself, the coupling between components would be too tight. We would like to achieve "loose coupling", and this is where twisted.python.components comes in.

First, we need to discuss Interfaces in more detail. As we mentioned earlier, an Interface is nothing more than a class which is used as a marker. Interfaces should be subclasses of `zope.interface.Interface`, and have a very odd look to python programmers not used to them:

```
from zope.interface import Interface
```

```
class IAmericanSocket(Interface):
    def voltage():
      """
      Return the voltage produced by this socket object, as an integer.
      """
```

Notice how it looks just like a regular class definition, other than inheriting from `Interface`? However, the method definitions inside the class block do not have any method body! Since Python does not have any native language-level support for Interfaces like Java does, this is what distinguishes an Interface definition from a Class.

Now that we have a defined Interface, we can talk about objects using terms like this: "The `AmericanSocket` class implements the `IAmericanSocket` interface" and "Please give me an object which adapts `UKSocket` to the `IAmericanSocket` interface". We can make *declarations* about what interfaces a certain class implements, and we can request adapters which implement a certain interface for a specific class.

Let's look at how we declare that a class implements an interface:

```
from zope.interface import implementer

@implementer(IAmericanSocket)
class AmericanSocket:
    def voltage(self):
        return 120
```

So, to declare that a class implements an interface, we simply decorate it with `zope.interface.implementer`.

Now, let's say we want to rewrite the `AdaptToAmericanSocket` class as a real adapter. In this case we also specify it as implementing `IAmericanSocket`:

```
from zope.interface import implementer

@implementer(IAmericanSocket)
class AdaptToAmericanSocket:
    def __init__(self, original):
        """
        Pass the original UKSocket object as original
        """
        self.original = original

    def voltage(self):
        return self.original.voltage() / 2
```

Notice how we placed the implements declaration on this adapter class. So far, we have not achieved anything by using components other than requiring us to type more. In order for components to be useful, we must use the *component registry*. Since `AdaptToAmericanSocket` implements `IAmericanSocket` and regulates the voltage of a `UKSocket` object, we can register `AdaptToAmericanSocket` as an `IAmericanSocket` adapter for the `UKSocket` class. It is easier to see how this is done in code than to describe it:

```
from zope.interface import Interface, implementer
from twisted.python import components

class IAmericanSocket(Interface):
    def voltage():
      """
      Return the voltage produced by this socket object, as an integer.
      """

@implementer(IAmericanSocket)
```

```
class AmericanSocket:
    def voltage(self):
        return 120

class UKSocket:
    def voltage(self):
        return 240

@implementer(IAmericanSocket)
class AdaptToAmericanSocket:
    def __init__(self, original):
        self.original = original

    def voltage(self):
        return self.original.voltage() / 2

components.registerAdapter(
    AdaptToAmericanSocket,
    UKSocket,
    IAmericanSocket)
```

Now, if we run this script in the interactive interpreter, we can discover a little more about how to use components. The first thing we can do is discover whether an object implements an interface or not:

```
>>> IAmericanSocket.implementedBy(AmericanSocket)
True
>>> IAmericanSocket.implementedBy(UKSocket)
False
>>> am = AmericanSocket()
>>> uk = UKSocket()
>>> IAmericanSocket.providedBy(am)
True
>>> IAmericanSocket.providedBy(uk)
False
```

As you can see, the `AmericanSocket` instance claims to implement `IAmericanSocket`, but the `UKSocket` does not. If we wanted to use the `HairDryer` with the `AmericanSocket`, we could know that it would be safe to do so by checking whether it implements `IAmericanSocket`. However, if we decide we want to use `HairDryer` with a `UKSocket` instance, we must *adapt* it to `IAmericanSocket` before doing so. We use the interface object to do this:

```
>>> IAmericanSocket(uk)
<__main__.AdaptToAmericanSocket instance at 0x1a5120>
```

When calling an interface with an object as an argument, the interface looks in the adapter registry for an adapter which implements the interface for the given instance's class. If it finds one, it constructs an instance of the Adapter class, passing the constructor the original instance, and returns it. Now the `HairDryer` can safely be used with the adapted `UKSocket` . But what happens if we attempt to adapt an object which already implements `IAmericanSocket`? We simply get back the original instance:

```
>>> IAmericanSocket(am)
<__main__.AmericanSocket instance at 0x36bff0>
```

So, we could write a new "smart" `HairDryer` which automatically looked up an adapter for the socket you tried to plug it into:

```
class HairDryer:
    def plug(self, socket):
        adapted = IAmericanSocket(socket)
        assert adapted.voltage() == 120, "BOOM"
        print("I was plugged in properly and am operating")
```

Now, if we create an instance of our new "smart" `HairDryer` and attempt to plug it in to various sockets, the `HairDryer` will adapt itself automatically depending on the type of socket it is plugged in to:

```
>>> am = AmericanSocket()
>>> uk = UKSocket()
>>> hd = HairDryer()
>>> hd.plug(am)
I was plugged in properly and am operating
>>> hd.plug(uk)
I was plugged in properly and am operating
```

Voila; the magic of components.

Components and Inheritance

If you inherit from a class which implements some interface, and your new subclass declares that it implements another interface, the implements will be inherited by default.

For example, pb.Root is a class which implements IPBRoot. This interface indicates that an object has remotely-invokable methods and can be used as the initial object served by a new Broker instance. It has an `implements` setting like:

```
from zope.interface import implementer

@implementer(IPBRoot)
class Root(Referenceable):
    pass
```

Suppose you have your own class which implements your `IMyInterface` interface:

```
from zope.interface import implementer, Interface

class IMyInterface(Interface):
    pass

@implementer(IMyInterface)
class MyThing:
    pass
```

Now if you want to make this class inherit from `pb.Root`, the interfaces code will automatically determine that it also implements `IPBRoot`:

```
from twisted.spread import pb
from zope.interface import implementer, Interface

class IMyInterface(Interface):
    pass

@implementer(IMyInterface)
```

```
class MyThing(pb.Root):
    pass
```

```
>>> from twisted.spread.flavors import IPBRoot
>>> IPBRoot.implementedBy(MyThing)
True
```

If you want `MyThing` to inherit from `pb.Root` but *not* implement `IPBRoot` like `pb.Root` does, use `@implementer_only`:

```
from twisted.spread import pb
from zope.interface import implementer_only, Interface

class IMyInterface(Interface):
    pass

@implementer_only(IMyInterface)
class MyThing(pb.Root):
    pass
```

```
>>> from twisted.spread.pb import IPBRoot
>>> IPBRoot.implementedBy(MyThing)
False
```

Cred: Pluggable Authentication

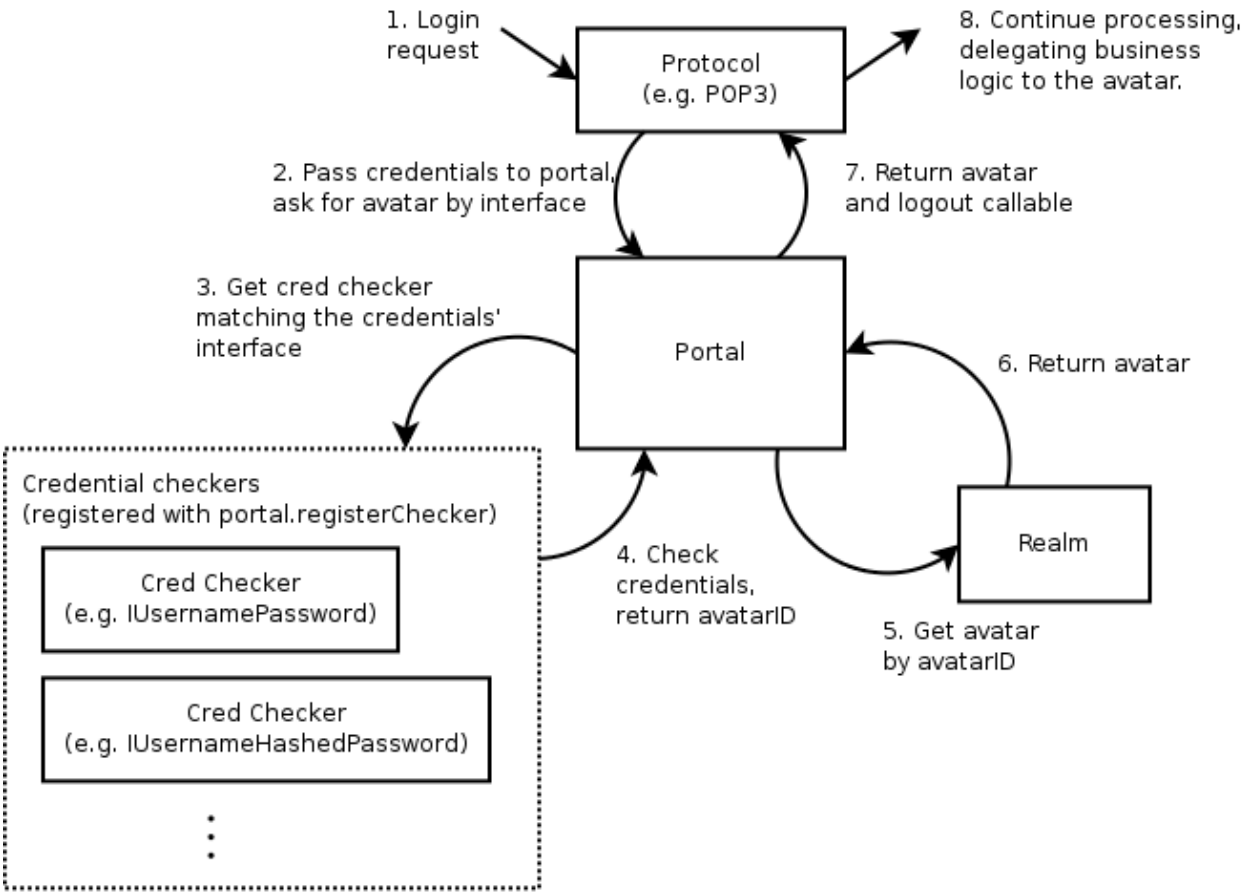
Goals

Cred is a pluggable authentication system for servers. It allows any number of network protocols to connect and authenticate to a system, and communicate to those aspects of the system which are meaningful to the specific protocol. For example, Twisted's POP3 support passes a "username and password" set of credentials to get back a mailbox for the specified email account. IMAP does the same, but retrieves a slightly different view of the same mailbox, enabling those features specific to IMAP which are not available in other mail protocols.

Cred is designed to allow both the backend implementation of the business logic - called the *avatar* - and the authentication database - called the *credential checker* - to be decided during deployment. For example, the same POP3 server should be able to authenticate against the local UNIX password database or an LDAP server without having to know anything about how or where mail is stored.

To sketch out how this works - a "Realm" corresponds to an application domain and is in charge of avatars, which are network-accessible business logic objects. To connect this to an authentication database, a top-level object called a Portal stores a realm, and a number of credential checkers. Something that wishes to log in, such as a Protocol , stores a reference to the portal. Login consists of passing credentials and a request interface (e.g. POP3's IMailbox) to the portal. The portal passes the credentials to the appropriate credential checker, which returns an avatar ID. The ID is passed to the realm, which returns the appropriate avatar. For a Portal that has a realm that creates mailbox objects and a credential checker that checks /etc/passwd, login consists of passing in a username/password and the IMailbox interface to the portal. The portal passes this to the /etc/passwd credential checker, gets back a avatar ID corresponding to an email account, passes that to the realm and gets back a mailbox object for that email account.

Putting all this together, here's how a login request will typically be processed:



Cred objects

The Portal

This is the core of login, the point of integration between all the objects in the cred system. There is one concrete implementation of Portal, and no interface - it does a very simple task. A Portal associates one (1) Realm with a collection of CredentialChecker instances. (More on those later.)

If you are writing a protocol that needs to authenticate against something, you will need a reference to a Portal, and to nothing else. This has only 2 methods -

- login `(credentials, mind, *interfaces)`

  The docstring is quite expansive (see twisted.cred.portal), but in brief, this is what you call when you need to call in order to connect a user to the system. Typically you only pass in one interface, and the mind is `None` . The interfaces are the possible interfaces the returned avatar is expected to implement, in order of preference. The result is a deferred which fires a tuple of:

  - interface the avatar implements (which was one of the interfaces passed in the `*interfaces` tuple)
  - an object that implements that interface (an avatar)
  - logout, a 0-argument callable which disconnects the connection that was established by this call to login

  The logout method has to be called when the avatar is logged out. For POP3 this means when the protocol is disconnected or logged out, etc..

- registerChecker `(checker, *credentialInterfaces)`

which adds a CredentialChecker to the portal. The optional list of interfaces are interfaces of credentials that the checker is able to check.

The CredentialChecker

This is an object implementing ICredentialsChecker which resolves some credentials to an avatar ID.

Whether the credentials are stored in an in-memory data structure, an Apache-style htaccess file, a UNIX password database, an SSH key database, or any other form, an implementation of `ICredentialsChecker` is how this data is connected to cred.

A credential checker stipulates some requirements of the credentials it can check by specifying a credentialInterfaces attribute, which is a list of interfaces. Credentials passed to its requestAvatarId method must implement one of those interfaces.

For the most part, these things will just check usernames and passwords and produce the username as the result, but hopefully we will be seeing some public-key, challenge-response, and certificate based credential checker mechanisms soon.

A credential checker should raise an error if it cannot authenticate the user, and return `twisted.cred.checkers.ANONYMOUS` for anonymous access.

The Credentials

Oddly enough, this represents some credentials that the user presents. Usually this will just be a small static blob of data, but in some cases it will actually be an object connected to a network protocol. For example, a username/password pair is static, but a challenge/response server is an active state-machine that will require several method calls in order to determine a result.

Twisted comes with a number of credentials interfaces and implementations in the twisted.cred.credentials module, such as IUsernamePassword and IUsernameHashedPassword .

The Realm

A realm is an interface which connects your universe of "business objects" to the authentication system.

IRealm is another one-method interface:

- requestAvatar `(avatarId, mind, *interfaces)`

  This method will typically be called from 'Portal.login'. The avatarId is the one returned by a CredentialChecker.

  Note: Note that `avatarId` must always be a string. In particular, do not use unicode strings. If internationalized support is needed, it is recommended to use UTF-8, and take care of decoding in the realm.

  The important thing to realize about this method is that if it is being called, *the user has already authenticated* . Therefore, if possible, the Realm should create a new user if one does not already exist whenever possible. Of course, sometimes this will be impossible without more information, and that is the case that the interfaces argument is for.

Since requestAvatar should be called from a Deferred callback, it may return a Deferred or a synchronous result.

The Avatar

An avatar is a business logic object for a specific user. For POP3, it's a mailbox, for a first-person-shooter it's the object that interacts with the game, the actor as it were. Avatars are specific to an application, and each avatar represents a single "user" .

The Mind

As mentioned before, the mind is usually `None` , so you can skip this bit if you want.

Masters of Perspective Broker already know this object as the ill-named "client object" . There is no "mind" class, or even interface, but it is an object which serves an important role - any notifications which are to be relayed to an authenticated client are passed through a 'mind'. In addition, it allows passing more information to the realm during login in addition to the avatar ID.

The name may seem rather unusual, but considering that a Mind is representative of the entity on the "other end" of a network connection that is both receiving updates and issuing commands, I believe it is appropriate.

Although many protocols will not use this, it serves an important role. It is provided as an argument both to the Portal and to the Realm, although a CredentialChecker should interact with a client program exclusively through a Credentials instance.

Unlike the original Perspective Broker "client object" , a Mind's implementation is most often dictated by the protocol that is connecting rather than the Realm. A Realm which requires a particular interface to issue notifications will need to wrap the Protocol's mind implementation with an adapter in order to get one that conforms to its expected interface - however, Perspective Broker will likely continue to use the model where the client object has a pre-specified remote interface.

(If you don't quite understand this, it's fine. It's hard to explain, and it's not used in simple usages of cred, so feel free to pass None until you find yourself requiring something like this.)

Responsibilities

Server protocol implementation

The protocol implementor should define the interface the avatar should implement, and design the protocol to have a portal attached. When a user logs in using the protocol, a credential object is created, passed to the portal, and an avatar with the appropriate interface is requested. When the user logs out or the protocol is disconnected, the avatar should be logged out.

The protocol designer should not hardcode how users are authenticated or the realm implemented. For example, a POP3 protocol implementation would require a portal whose realm returns avatars implementing IMailbox and whose credential checker accepts username/password credentials, but that is all. Here's a sketch of how the code might look - note that USER and PASS are the protocol commands used to login, and the DELE command can only be used after you are logged in:

`pop3_server.py`

```
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from zope.interface import Interface

from twisted.protocols import basic
from twisted.python import log
from twisted.cred import credentials, error
```

```
from twisted.internet import defer

class IMailbox(Interface):
    """
    Interface specification for mailbox.
    """
    def deleteMessage(index):
        pass


class POP3(basic.LineReceiver):
    # ...
    def __init__(self, portal):
        self.portal = portal

    def do_DELE(self, i):
        # uses self.mbox, which is set after login
        i = int(i)-1
        self.mbox.deleteMessage(i)
        self.successResponse()

    def do_USER(self, user):
        self._userIs = user
        self.successResponse('USER accepted, send PASS')

    def do_PASS(self, password):
        if self._userIs is None:
            self.failResponse("USER required before PASS")
            return
        user = self._userIs
        self._userIs = None
        d = defer.maybeDeferred(self.authenticateUserPASS, user, password)
        d.addCallback(self._cbMailbox, user)

    def authenticateUserPASS(self, user, password):
        if self.portal is not None:
            return self.portal.login(
                credentials.UsernamePassword(user, password),
                None,
                IMailbox
            )
        raise error.UnauthorizedLogin()

    def _cbMailbox(self, ial, user):
        interface, avatar, logout = ial

        if interface is not IMailbox:
            self.failResponse('Authentication failed')
            log.err("_cbMailbox() called with an interface other than IMailbox")
            return

        self.mbox = avatar
        self._onLogout = logout
        self.successResponse('Authentication succeeded')
        log.msg("Authenticated login for " + user)
```

Application implementation

The application developer can implement realms and credential checkers. For example, they might implement a realm that returns IMailbox implementing avatars, using MySQL for storage, or perhaps a credential checker that uses LDAP for authentication. In the following example, the Realm for a simple remote object service (using Twisted's Perspective Broker protocol) is implemented:

```
from zope.interface import implementer

from twisted.spread import pb
from twisted.cred.portal import IRealm

class SimplePerspective(pb.Avatar):

    def perspective_echo(self, text):
        print('echoing',text)
        return text

    def logout(self):
        print(self, "logged out")


@implementer(IRealm)
class SimpleRealm:

    def requestAvatar(self, avatarId, mind, *interfaces):
        if pb.IPerspective in interfaces:
            avatar = SimplePerspective()
            return pb.IPerspective, avatar, avatar.logout
        else:
            raise NotImplementedError("no interface")
```

Deployment

Deployment involves tying together a protocol, an appropriate realm and a credential checker. For example, a POP3 server can be constructed by attaching to it a portal that wraps the MySQL-based realm and an /etc/passwd credential checker, or perhaps the LDAP credential checker if that is more useful. The following example shows how the SimpleRealm in the previous example is deployed using an in-memory credential checker:

```
from twisted.spread import pb
from twisted.internet import reactor
from twisted.cred.portal import Portal
from twisted.cred.checkers import InMemoryUsernamePasswordDatabaseDontUse

portal = Portal(SimpleRealm())
checker = InMemoryUsernamePasswordDatabaseDontUse()
checker.addUser("guest", "password")
portal.registerChecker(checker)
reactor.listenTCP(9986, pb.PBServerFactory(portal))
reactor.run()
```

Cred plugins

Authentication with cred plugins

Cred offers a plugin architecture for authentication methods. The primary API for this architecture is the command-line; the plugins are meant to be specified by the end-user when deploying a TAP (twistd plugin).

For more information on writing a twistd plugin and using cred plugins for your application, please refer to the *Writing a twistd plugin* document.

Building a cred plugin

To build a plugin for cred, you should first define an `authType` , a short one-word string that defines your plugin to the command-line. Once you have this, the convention is to create a file named `myapp_plugins.py` in the twisted.plugins module path.

Below is an example file structure for an application that defines such a plugin:

- MyApplication/
  - setup.py
  - myapp/
    - __init__.py
    - cred.py
    - server.py
  - twisted/
    - plugins/
      - myapp_plugins.py

Once you have created this structure within your application, you can create the code for your cred plugin by building a factory class which implements ICheckerFactory . These factory classes should not consist of a tremendous amount of code. Most of the real application logic should reside in the cred checker itself. (For help on building those, scroll up.)

The core purpose of the CheckerFactory is to translate an `argstring` , which is passed on the command line, into a suitable set of initialization parameters for a Checker class. In most cases this should be little more than constructing a dictionary or a tuple of arguments, then passing them along to a new checker instance.

```
from zope.interface import implementer

from twisted import plugin
from twisted.cred.strcred import ICheckerFactory
from myapp.cred import SpecialChecker

# The class needs to implement both of these interfaces
# for the plugin system to find our factory.
@implementer(ICheckerFactory, plugin.IPlugin)
class SpecialCheckerFactory(object):
    """
    A checker factory for a specialized (fictional) API.
    """
    # This tells AuthOptionsMixin how to find this factory.
    authType = "special"

    # This is a one-line explanation of what arguments, if any,
    # your particular cred plugin requires at the command-line.
```

```
    argStringFormat = "A colon-separated key=value list."

    # This help text can be multiple lines. It will be displayed
    # when someone uses the "--help-auth-type special" command.
    authHelp = """Some help text goes here ..."""

    # This will be called once per command-line.
    def generateChecker(self, argstring=""):
        argdict = dict((x.split('=') for x in argstring.split(':')))
        return SpecialChecker(**argdict)

# We need to instantiate our class for the plugin to work.
theSpecialCheckerFactory = SpecialCheckerFactory()
```

For more information on how your plugin can be used in your application (and by other application developers), please see the *Writing a twistd plugin* document.

Conclusion

After reading through this tutorial, you should be able to

- Understand how the cred architecture applies to your application
- Integrate your application with cred's object model
- Deploy an application that uses cred for authentication
- Allow your users to use command-line authentication plugins

The Twisted Plugin System

The purpose of this guide is to describe the preferred way to write extensible Twisted applications (and consequently, also to describe how to extend applications written in such a way). This extensibility is achieved through the definition of one or more APIs and a mechanism for collecting code plugins which implement this API to provide some additional functionality. At the base of this system is the twisted.plugin module.

Making an application extensible using the plugin system has several strong advantages over other techniques:

- It allows third-party developers to easily enhance your software in a way that is loosely coupled: only the plugin API is required to remain stable.
- It allows new plugins to be discovered flexibly. For example, plugins can be loaded and saved when a program is first run, or re-discovered each time the program starts up, or they can be polled for repeatedly at runtime (allowing the discovery of new plugins installed after the program has started).

Writing Extensible Programs

Taking advantage of twisted.plugin is a two step process:

1. Define an interface which plugins will be required to implement. This is done using the zope.interface package in the same way one would define an interface for any other purpose.

   A convention for defining interfaces is do so in a file named like *ProjectName/projectname/iprojectname.py* . The rest of this document will follow that convention: consider the following interface definition be in `Matsim/matsim/imatsim.py` , an interface definition module for a hypothetical material simulation package.

2. At one or more places in your program, invoke twisted.plugin.getPlugins and iterate over its result.

As an example of the first step, consider the following interface definition for a physical modelling system.

```
from zope.interface import Interface, Attribute

class IMaterial(Interface):
    """
    An object with specific physical properties
    """
    def yieldStress(temperature):
        """
        Returns the pressure this material can support without
        fracturing at the given temperature.

        @type temperature: C{float}
        @param temperature: Kelvins

        @rtype: C{float}
        @return: Pascals
        """

    dielectricConstant = Attribute("""
        @type dielectricConstant: C{complex}
        @ivar dielectricConstant: The relative permittivity, with the
        real part giving reflective surface properties and the
        imaginary part giving the radio absorption coefficient.
        """)
```

In another module, we might have a function that operates on objects providing the `IMaterial` interface:

```
def displayMaterial(m):
    print('A material with yield stress %s at 500 K' % (m.yieldStress(500),))
    print('Also a dielectric constant of %s.' % (m.dielectricConstant,))
```

The last piece of required code is that which collects `IMaterial` providers and passes them to the `displayMaterial` function.

```
from twisted.plugin import getPlugins
from matsim import imatsim

def displayAllKnownMaterials():
    for material in getPlugins(imatsim.IMaterial):
        displayMaterial(material)
```

Third party developers may now contribute different materials to be used by this modelling system by implementing one or more plugins for the `IMaterial` interface.

Extending an Existing Program

The above code demonstrates how an extensible program might be written using Twisted's plugin system. How do we write plugins for it, though? Essentially, we create objects which provide the required interface and then make them available at a particular location. Consider the following example.

```
from zope.interface import implementer
from twisted.plugin import IPlugin
from matsim import imatsim

@implementer(IPlugin, imatsim.IMaterial)
```

```
class SimpleMaterial(object):
    def __init__(self, yieldStressFactor, dielectricConstant):
        self._yieldStressFactor = yieldStressFactor
        self.dielectricConstant = dielectricConstant

    def yieldStress(self, temperature):
        return self._yieldStressFactor * temperature

steelPlate = SimpleMaterial(2.06842719e11, 2.7 + 0.2j)
brassPlate = SimpleMaterial(1.03421359e11, 1.4 + 0.5j)
```

`steelPlate` and `brassPlate` now provide both IPlugin and `IMaterial` . All that remains is to make this module available at an appropriate location. For this, there are two options. The first of these is primarily useful during development: if a directory which has been added to `sys.path` (typically by adding it to the `PYTHONPATH` environment variable) contains a *directory* named `twisted/plugins/` , each `.py` file in that directory will be loaded as a source of plugins. This directory *must not* be a Python package: including `__init__.py` will cause the directory to be skipped and no plugins loaded from it. Second, each module in the installed version of Twisted's `twisted.plugins` package will also be loaded as a source of plugins.

Once this plugin is installed in one of these two ways, `displayAllKnownMaterials` can be run and we will see two pairs of output: one for a steel plate and one for a brass plate.

Alternate Plugin Packages

getPlugins takes one additional argument not mentioned above. If passed in, the 2nd argument should be a module or package to be used instead of `twisted.plugins` as the plugin meta-package. If you are writing a plugin for a Twisted interface, you should never need to pass this argument. However, if you have developed an interface of your own, you may want to mandate that plugins for it are installed in your own plugins package, rather than in Twisted's.

You may want to support `yourproject/plugins/` directories for ease of development. To do so, you should make `yourproject/plugins/__init__.py` contain at least the following lines.

```
from twisted.plugin import pluginPackagePaths
__path__.extend(pluginPackagePaths(__name__))
__all__ = []
```

The key behavior here is that interfaces are essentially paired with a particular plugin package. If plugins are installed in a different package than the one the code which relies on the interface they provide, they will not be found when the application goes to load them.

Plugin Caching

In the course of using the Twisted plugin system, you may notice `dropin.cache` files appearing at various locations. These files are used to cache information about what plugins are present in the directory which contains them. At times, this cached information may become out of date. Twisted uses the mtimes of various files involved in the plugin system to determine when this cache may have become invalid. Twisted will try to re-write the cache each time it tries to use it but finds it out of date.

For a site-wide install, it may not (indeed, should not) be possible for applications running as normal users to rewrite the cache file. While these applications will still run and find correct plugin information, they may run more slowly than they would if the cache was up to date, and they may also report exceptions if certain plugins have been removed but which the cache still references. For these reasons, when installing or removing software which provides Twisted plugins, the site administrator should be sure the cache is regenerated. Well-behaved package managers for such software should take this task upon themselves, since it is trivially automatable. The canonical way to regenerate the cache is to run the following Python code:

```
from twisted.plugin import IPlugin, getPlugins
list(getPlugins(IPlugin))
```

As mentioned, it is normal for exceptions to be raised **once** here if plugins have been removed.

Further Reading

- *Components: Interfaces and Adapters*

The Basics

Application

Twisted programs usually work with twisted.application.service.Application. This class usually holds all persistent configuration of a running server, such as:

- ports to bind to,
- places where connections to must be kept or attempted,
- periodic actions to do,
- and almost everything else to do with your Application.

It is the root object in a tree of services implementing twisted.application.service.IService.

Other howtos describe how to write custom code for `Applications`, but this one describes how to use already written code (which can be part of Twisted or from a third-party Twisted plugin developer). The Twisted distribution comes with an important tool to deal with `Applications`: `twistd(1)`.

`Applications` are just Python objects, which can be created and manipulated in the same ways as any other object.

twistd

The Twisted Daemon is a program that knows how to run Applications. Strictly speaking, `twistd` is not necessary. Fetching the application, getting the `IService` component, calling `startService()`, scheduling `stopService()` when the reactor shuts down, and then calling `reactor.run()` could be done manually.

However, `twistd` supplies many options which are highly useful for program set up:

- choosing a reactor (for more on reactors, see *Choosing a Reactor*),
- logging configuration (see the *logger* documentation for more),
- daemonizing (forking to the background),
- and *more*.

`twistd` supports all Applications mentioned above – and an additional one. Sometimes it is convenient to write the code for building a class in straight Python. One big source of such Python files is the *examples* directory. When a straight Python file which defines an `Application` object called `application` is used, use the `-y` option.

When `twistd` runs, it records its process id in a `twistd.pid` file (this can be configured via a command line switch). In order to shutdown the `twistd` process, kill that pid. The usual way to do this would be:

```
kill `cat twistd.pid`
```

To prevent `twistd` from daemonizing, you can pass it the `--no-daemon` option (or `-n`, in conjunction with other short options).

As always, the gory details are in the manual page.

Using the Twisted Application Framework

Introduction

Audience

The target audience of this document is a Twisted user who wants to deploy a significant amount of Twisted code in a re-usable, standard and easily configurable fashion. A Twisted user who wishes to use the Application framework needs to be familiar with developing Twisted *servers* and/or *clients*.

Goals

- To introduce the Twisted Application infrastructure.
- To explain how to deploy your Twisted application using `.tac` files and `twistd`.
- To outline the existing Twisted services.

Overview

The Twisted Application infrastructure takes care of running and stopping your application. Using this infrastructure frees you from from having to write a large amount of boilerplate code by hooking your application into existing tools that manage daemonization, logging, *choosing a reactor* and more.

The major tool that manages Twisted applications is a command-line utility called `twistd`. `twistd` is cross platform, and is the recommended tool for running Twisted applications.

The core component of the Twisted Application infrastructure is the twisted.application.service.Application object – an object which represents your application. However, Application doesn't provide anything that you'd want to manipulate directly. Instead, Application acts as a container of any "Services" (objects implementing IService) that your application provides. Most of your interaction with the Application infrastructure will be done through Services.

By "Service", we mean anything in your application that can be started and stopped. Typical services include web servers, FTP servers and SSH clients. Your Application object can contain many services, and can even contain structured hierarchies of Services using MultiService or your own custom IServiceCollection implementations. You will most likely want to use these to manage Services which are dependent on other Services. For example, a proxying Twisted application might want its server Service to only start up after the associated Client service.

An IService has two basic methods, `startService()` which is used to start the service, and `stopService()` which is used to stop the service. The latter can return a Deferred, indicating service shutdown is not over until the result fires. For example:

```
from twisted.internet import reactor
from twisted.application import service
from somemodule import EchoFactory

class EchoService(service.Service):
    def __init__(self, portNum):
        self.portNum = portNum
```

```
    def startService(self):
        self._port = reactor.listenTCP(self.portNum, EchoFactory())

    def stopService(self):
        return self._port.stopListening()
```

See *Writing Servers* for an explanation of `EchoFactory` and `listenTCP`.

Using Services and Application

twistd and tac

To handle start-up and configuration of your Twisted application, the Twisted Application infrastructure uses `.tac` files. `.tac` are Python files which configure an Application object and assign this object to the top-level variable "`application`".

The following is a simple example of a `.tac` file:

`service.tac`

```
# You can run this .tac file directly with:
#    twistd -ny service.tac

"""
This is an example .tac file which starts a webserver on port 8080 and
serves files from the current working directory.

The important part of this, the part that makes it a .tac file, is
the final root-level section, which sets up the object called 'application'
which twistd will look for
"""

import os
from twisted.application import service, internet
from twisted.web import static, server

def getWebService():
    """
    Return a service suitable for creating an application object.

    This service is a simple web server that serves files on port 8080 from
    underneath the current working directory.
    """
    # create a resource to serve static files
    fileServer = server.Site(static.File(os.getcwd()))
    return internet.TCPServer(8080, fileServer)

# this is the core part of any tac file, the creation of the root-level
# application object
application = service.Application("Demo application")

# attach the service to its parent application
service = getWebService()
service.setServiceParent(application)
```

`twistd` is a program that runs Twisted applications using a `.tac` file. In its most simple form, it takes a single argument `-y` and a tac file name. For example, you can run the above server with the command `twistd -y`

`service.tac`.

By default, `twistd` daemonizes and logs to a file called `twistd.log`. More usually, when debugging, you will want your application to run in the foreground and log to the command line. To run the above file like this, use the command `twistd -noy service.tac`.

For more information, see the `twistd` man page.

Customizing `twistd` logging

`twistd` logging can be customized using the command line. This requires that a *log observer factory* be importable. Given a file named `my.py` with the code:

```
from twisted.logger import textFileLogObserver

def logger():
    return textFileLogObserver(open("/tmp/my.log", "w"))
```

Invoking `twistd --logger my.logger ...` will log to a file named `/tmp/my.log` (this simple example could easily be replaced with use of the `--logfile` parameter to twistd).

Alternatively, the logging behavior can be customized through an API accessible from `.tac` files. The ILogObserver component can be set on an Application in order to customize the default log observer that `twistd` will use.

Here is an example of how to use DailyLogFile, which rotates the log once per day.

```
from twisted.application.service import Application
from twisted.logger import ILogObserver, textFileLogObserver
from twisted.python.logfile import DailyLogFile

application = Application("myapp")
logfile = DailyLogFile("my.log", "/tmp")
application.setComponent(ILogObserver, textFileLogObserver(logfile))
```

Invoking `twistd -y my.tac` will create a log file at `/tmp/my.log`.

Services provided by Twisted

Twisted also provides pre-written IService implementations for common cases like listening on a TCP port, in the twisted.application.internet module. Here's a simple example of constructing a service that runs an echo server on TCP port 7001:

```
from twisted.application import internet, service
from somemodule import EchoFactory

port = 7001
factory = EchoFactory()

echoService = internet.TCPServer(port, factory) # create the service
```

Each of these services (except TimerService) has a corresponding "connect" or "listen" method on the reactor, and the constructors for the services take the same arguments as the reactor methods. The "connect" methods are for clients and the "listen" methods are for servers. For example, `TCPServer` corresponds to `reactor.listenTCP` and `TCPClient` corresponds to `reactor.connectTCP`.

`TCPServer`

`TCPClient`

Services which allow you to make connections and listen for connections on TCP ports.

- listenTCP
- connectTCP

`UNIXServer`

`UNIXClient`

Services which listen and make connections over UNIX sockets.

- listenUNIX
- connectUNIX

`SSLServer`

`SSLClient`

Services which allow you to make SSL connections and run SSL servers.

- listenSSL
- connectSSL

`UDPServer`

A service which allows you to send and receive data over UDP.

- listenUDP

See also the *UDP documentation*.

`UNIXDatagramServer`

`UNIXDatagramClient`

Services which send and receive data over UNIX datagram sockets.

- listenUNIXDatagram
- connectUNIXDatagram

`MulticastServer`

A server for UDP socket methods that support multicast.

- listenMulticast

`TimerService`

A service to periodically call a function.

- TimerService

Service Collection

IServiceCollection objects contain IService objects. IService objects can be added to IServiceCollection by calling setServiceParent and detached by using disownServiceParent.

The standard implementation of IServiceCollection is MultiService, which also implements IService. MultiService is useful for creating a new Service which combines two or more existing Services. For example, you could create a DNS Service as a MultiService which has a TCP and a UDP Service as children.

```
from twisted.application import internet, service
from twisted.names import server, dns, hosts

port = 53

# Create a MultiService, and hook up a TCPServer and a UDPServer to it as
# children.
dnsService = service.MultiService()
hostsResolver = hosts.Resolver('/etc/hosts')
tcpFactory = server.DNSServerFactory([hostsResolver])
internet.TCPServer(port, tcpFactory).setServiceParent(dnsService)
udpFactory = dns.DNSDatagramProtocol(tcpFactory)
internet.UDPServer(port, udpFactory).setServiceParent(dnsService)

# Create an application as normal
application = service.Application("DNSExample")

# Connect our MultiService to the application, just like a normal service.
dnsService.setServiceParent(application)
```

Writing a twistd Plugin

This document describes adding subcommands to the `twistd` command, as a way to facilitate the deployment of your applications. *(This feature was added in Twisted 2.5)*

The target audience of this document are those that have developed a Twisted application which needs a command line-based deployment mechanism.

There are a few prerequisites to understanding this document:

- A basic understanding of the Twisted Plugin System (i.e., the twisted.plugin module) is necessary, however, step-by-step instructions will be given. Reading *The Twisted Plugin System* is recommended, in particular the "Extending an Existing Program" section.
- The *Application* infrastructure is used in `twistd` plugins; in particular, you should know how to expose your program's functionality as a Service.
- In order to parse command line arguments, the `twistd` plugin mechanism relies on `twisted.python.usage` , which is documented in *Using usage.Options* .

Goals

After reading this document, the reader should be able to expose their Service-using application as a subcommand of `twistd` , taking into consideration whatever was passed on the command line.

Alternatives to twistd plugins

The major alternative to the twistd plugin mechanism is the `.tac` file, which is a simple script to be used with the twistd `-y/--python` parameter. The twistd plugin mechanism exists to offer a more extensible command-line-driven interface to your application. For more information on `.tac` files, see the document *Using the Twisted Application Framework* .

Creating the plugin

The following directory structure is assumed of your project:

- **MyProject** - Top level directory
  - **myproject** - Python package
    - **__init__.py**

During development of your project, Twisted plugins can be loaded from a special directory in your project, assuming your top level directory ends up in sys.path. Create a directory named `twisted` containing a directory named `plugins` , and add a file named `myproject_plugin.py` to it. This file will contain your plugin. Note that you should *not* add any __init__.py files to this directory structure, and the plugin file should *not* be named `myproject.py` (because that would conflict with your project's module name).

In this file, define an object which *provides* the interfaces twisted.plugin.IPlugin and twisted.application.service.IServiceMaker .

The `tapname` attribute of your IServiceMaker provider will be used as the subcommand name in a command like `twistd [subcommand] [args...]` , and the `options` attribute (which should be a usage.Options subclass) will be used to parse the given args.

```
from zope.interface import implementer

from twisted.python import usage
from twisted.plugin import IPlugin
from twisted.application.service import IServiceMaker
from twisted.application import internet

from myproject import MyFactory


class Options(usage.Options):
    optParameters = [["port", "p", 1235, "The port number to listen on."]]


@implementer(IServiceMaker, IPlugin)
class MyServiceMaker(object):
    tapname = "myproject"
    description = "Run this! It'll make your dog happy."
    options = Options

    def makeService(self, options):
        """
        Construct a TCPServer from a factory defined in myproject.
        """
        return internet.TCPServer(int(options["port"]), MyFactory())


# Now construct an object which *provides* the relevant interfaces
# The name of this variable is irrelevant, as long as there is *some*
# name bound to a provider of IPlugin and IServiceMaker.

serviceMaker = MyServiceMaker()
```

Now running `twistd --help` should print `myproject` in the list of available subcommands, followed by the description that we specified in the plugin. `twistd -n myproject` would, assuming we defined a `MyFactory` factory inside `myproject` , start a listening server on port 1235 with that factory.

Using cred with your TAP

Twisted ships with a robust authentication framework to use with your application. If your server needs authentication functionality, and you haven't read about *twisted.cred* yet, read up on it first.

If you are building a twistd plugin and you want to support a wide variety of authentication patterns, Twisted provides an easy-to-use mixin for your Options subclass: strcred.AuthOptionMixin . The following code is an example of using this mixin:

```
from twisted.cred import credentials, portal, strcred
from twisted.python import usage
from twisted.plugin import IPlugin
from twisted.application.service import IServiceMaker
from myserver import myservice

class ServerOptions(usage.Options, strcred.AuthOptionMixin):
    # This part is optional; it tells AuthOptionMixin what
    # kinds of credential interfaces the user can give us.
    supportedInterfaces = (credentials.IUsernamePassword,)

    optParameters = [
        ["port", "p", 1234, "Server port number"],
        ["host", "h", "localhost", "Server hostname"]]

@implementer(IServiceMaker, IPlugin)
class MyServerServiceMaker(object):
    tapname = "myserver"
    description = "This server does nothing productive."
    options = ServerOptions

    def makeService(self, options):
        """Construct a service object."""
        # The realm is a custom object that your server defines.
        realm = myservice.MyServerRealm(options["host"])

        # The portal is something Cred can provide, as long as
        # you have a list of checkers that you'll support. This
        # list is provided my AuthOptionMixin.
        portal = portal.Portal(realm, options["credCheckers"])

        # OR, if you know you might get multiple interfaces, and
        # only want to give your application one of them, you
        # also have that option with AuthOptionMixin:
        interface = credentials.IUsernamePassword
        portal = portal.Portal(realm, options["credInterfaces"][interface])

        # The protocol factory is, like the realm, something you implement.
        factory = myservice.ServerFactory(realm, portal)

        # Finally, return a service that will listen for connections.
        return internet.TCPServer(int(options["port"]), factory)


# As in our example above, we have to construct an object that
# provides the IPlugin and IServiceMaker interfaces.

serviceMaker = MyServerServiceMaker()
```

Now that you have your TAP configured to support any authentication we can throw at it, you're ready to use it. Here

is an example of starting your server using the /etc/passwd file for authentication. (Clearly, this won't work on servers with shadow passwords.)

```
$ twistd myserver --auth passwd:/etc/passwd
```

For a full list of cred plugins supported, see twisted.plugins , or use the command-line help:

```
$ twistd myserver --help-auth
$ twistd myserver --help-auth-type passwd
```

Conclusion

You should now be able to

- Create a twistd plugin
- Incorporate authentication into your plugin
- Use it from your development environment
- Install it correctly and use it in deployment

Deploying Twisted with systemd

Introduction

In this tutorial you will learn how to start a Twisted service using `systemd`. You will also learn how to start the service using `socket activation`.

Note: The examples in this tutorial demonstrate how to launch a Twisted web server, but the same techniques apply to any Twisted service.

Prerequisites

Twisted

You will need a version of Twisted >= 12.2 for the socket activation section of this tutorial.

This tutorial was written on a Fedora 18 Linux operating system with a system wide installation of Twisted and Twisted Web.

If you have installed Twisted locally eg in your home directory or in a virtualenv, you will need to modify the paths in some of the following examples.

Test your Twisted installation by starting a `twistd web` server on TCP port 8080 with the following command:

```
$ twistd --nodaemon web --port 8080 --path /srv/www/www.example.com/static
2013-01-28 13:21:35+0000 [-] Log opened.
2013-01-28 13:21:35+0000 [-] twistd 12.3.0 (/usr/bin/python 2.7.3) starting␣
↪up.
2013-01-28 13:21:35+0000 [-] reactor class: twisted.internet.epollreactor.
↪EPollReactor.
2013-01-28 13:21:35+0000 [-] Site starting on 8080
2013-01-28 13:21:35+0000 [-] Starting factory <twisted.web.server.Site␣
↪instance at 0x7f57eb66efc8>
```

This assumes that you have the following static web page in the following directory structure:

```
# tree /srv/
/srv/
- www
    - www.example.com
        - static
            - index.html
```

```
<!doctype html>
<html lang=en>
  <head>
    <meta charset=utf-8>
    <title>Example Site</title>
  </head>
  <body>
    <h1>Example Site</h1>
  </body>
</html>
```

Now try connecting to http://localhost:8080 in your web browser.

If you do not see your web page or if `twistd` didn't start, you should investigate and fix the problem before continuing.

Basic Systemd Service Configuration

The essential configuration file for a `systemd` service is the service file.

Later in this tutorial, you will learn about some other types of configuration file, which are used to control when and how your service is started.

But we will begin by configuring `systemd` to start a Twisted web server immediately on system boot.

Create a systemd.service file

Create the service file at `/etc/systemd/system/www.example.com.service` with the following content:

`/etc/systemd/system/www.example.com.service`

```
[Unit]
Description=Example Web Server

[Service]
ExecStart=/usr/bin/twistd \
    --nodaemon \
    --pidfile= \
    web --port 8080 --path .

WorkingDirectory=/srv/www/www.example.com/static

User=nobody
Group=nobody

Restart=always
```

```
[Install]
WantedBy=multi-user.target
```

This configuration file contains the following note worthy directives:

ExecStart

Always include the full path to `twistd` in case you have multiple versions installed.

The `--nodaemon` flag makes `twistd` run in the foreground. Systemd works best with child processes that remain in the foreground.

The `--pidfile=` flag prevents `twistd` from writing a pidfile. A pidfile is not necessary when Twisted runs as a foreground process.

The `--path` flag specifies the location of the website files. In this example we use "." which makes `twistd` serve files from its current working directory (see below).

WorkingDirectory

Systemd can configure the working environment of its child processes.

In this example the working directory of `twistd` is set to that of the static website.

User / Group

Systemd can also control the effective user and group of its child processes.

This example uses an un-privileged user "nobody" and un-privileged group "nobody".

This is an important security measure which ensures that the Twisted sub-process can not access restricted areas of the file system.

Restart

Systemd can automatically restart a child process if it exits or crashes unexpectedly.

In this example the `Restart` option is set to `always`, which ensures that `twistd` will be restarted under all circumstances.

WantedBy

Systemd service dependencies are controlled by `WantedBy` and `RequiredBy` directives in the `[Install]` section of configuration file.

The special multi-user.target is used in this example so that `systemd` starts the `twistd web` service when it reaches the multi-user stage of the boot sequence.

There are many more service directives which are documented in the systemd.directives man page.

Reload `systemd`

```
$ sudo systemctl daemon-reload
```

This forces `systemd` to read the new configuration file.

Always run `systemctl daemon-reload` after changing any of the `systemd` configuration files.

Start the service

```
$ sudo systemctl start www.example.com
```

`twistd` should now be running and listening on TCP port 8080. You can verify this using the `systemctl status` command. eg

```
$ systemctl status www.example.com.service
www.example.com.service - Example Web Server
          Loaded: loaded (/etc/systemd/system/www.example.com.service; enabled)
          Active: active (running) since Mon 2013-01-28 16:16:26 GMT; 1s ago
        Main PID: 10695 (twistd)
          CGroup: name=systemd:/system/www.example.com.service
                  -10695 /usr/bin/python /usr/bin/twistd --nodaemon --pidfile= web --
↪port 8080 --path .

Jan 28 16:16:26 zorin.lan systemd[1]: Starting Example Web Server...
Jan 28 16:16:26 zorin.lan systemd[1]: Started Example Web Server.
Jan 28 16:16:26 zorin.lan twistd[10695]: 2013-01-28 16:16:26+0000 [-] Log opened.
Jan 28 16:16:26 zorin.lan twistd[10695]: 2013-01-28 16:16:26+0000 [-] twistd 12.1.0 (/
↪usr/bin/python 2.7.3) starting up.
Jan 28 16:16:26 zorin.lan twistd[10695]: 2013-01-28 16:16:26+0000 [-] reactor class:␣
↪twisted.internet.epollreactor.EPollReactor.
Jan 28 16:16:26 zorin.lan twistd[10695]: 2013-01-28 16:16:26+0000 [-] Site starting␣
↪on 8080
Jan 28 16:16:26 zorin.lan twistd[10695]: 2013-01-28 16:16:26+0000 [-] Starting␣
↪factory <twisted.web.server.Site instance at 0x159b758>
```

The `systemctl status` command is convenient because it shows you both the current status of the service and a short log of the service output.

This is especially useful for debugging and diagnosing service startup problems.

The `twistd` subprocess will log messages to `stderr` and `systemd` will log these messages to syslog. You can verify this by monitoring the syslog messages or by using the new `journalctl` tool in Fedora.

See the systemctl man page for details of other `systemctl` command line options.

Enable the service

We've seen how to start the service manually, but now we need to "enable" it so that it starts automatically at boot time.

Enable the service with the following command:

```
$ sudo systemctl enable www.example.com.service
ln -s '/etc/systemd/system/www.example.com.service' '/etc/systemd/system/multi-user.
↪target.wants/www.example.com.service'
```

This creates a symlink to the service file in the `multi-user.target.wants` directory.

The Twisted web server will now be started automatically at boot time.

The `multi-user.target` is an example of a "special" systemd unit. Later in this tutorial you will learn how to use another special unit - the `sockets.target`.

Test that the service is automatically restarted

The `Restart=always` option in the `systemd.service` file ensures that `systemd` will restart the `twistd` process if and when it exits unexpectedly.

You can read about other `Restart` options in the systemd.service man page.

Try killing the `twistd` process and then checking its status again:

```
$ sudo kill 12543

$ systemctl status www.example.com.service
www.example.com.service - Example Web Server
          Loaded: loaded (/etc/systemd/system/www.example.com.service; disabled)
          Active: active (running) since Mon 2013-01-28 17:47:37 GMT; 1s ago
        Main PID: 12611 (twistd)
```

The "Active" time stamp shows that the `twistd` process was restarted within 1 second.

Now stop the service before you proceed to the next section.

```
$ sudo systemctl stop www.example.com.service

$ systemctl status www.example.com.service
www.example.com.service - Example Web Server
          Loaded: loaded (/etc/systemd/system/www.example.com.service; enabled)
          Active: inactive (dead) since Mon 2013-01-28 16:51:12 GMT; 1s ago
         Process: 10695 ExecStart=/usr/bin/twistd --nodaemon --pidfile= web --port␣
↪8080 --path . (code=exited, status=0/SUCCESS)
```

Socket Activation

First you need to understand what "socket activation" is. This extract from the systemd daemon man page explains it quite clearly.

> In a socket-based activation scheme the creation and binding of the listening socket as primary communication channel of daemons to local (and sometimes remote) clients is moved out of the daemon code and into the init system.
>
> Based on per-daemon configuration the init system installs the sockets and then hands them off to the spawned process as soon as the respective daemon is to be started.
>
> Optionally activation of the service can be delayed until the first inbound traffic arrives at the socket, to implement on-demand activation of daemons.
>
> However, the primary advantage of this scheme is that all providers and all consumers of the sockets can be started in parallel as soon as all sockets are established.
>
> In addition to that daemons can be restarted with losing only a minimal number of client transactions or even any client request at all (the latter is particularly true for state-less protocols, such as DNS or syslog), because the socket stays bound and accessible during the restart, and all requests are queued while the daemon cannot process them.

Another benefit of socket activation is that `systemd` can listen on privileged ports and start Twisted with privileges already dropped. This allows a Twisted service to be configured and restarted by a non-root user.

Twisted (since version 12.2) includes a systemd endpoint API and a corresponding string ports syntax which allows a Twisted service to inherit a listening socket from `systemd`.

The following example builds on the previous example, demonstrating how to enable socket activation for a simple Twisted web server.

Note: Before continuing, stop the previous example service with the following command:

```
$ sudo systemctl stop www.example.com.service
```

Create a systemd.socket file

Create the systemd.socket file at `/etc/systemd/system/www.example.com.socket` with the following content:

`/etc/systemd/system/www.example.com.socket`

```
[Socket]
ListenStream=0.0.0.0:80

[Install]
WantedBy=sockets.target
```

This configuration file contains the following important directives:

ListenStream=0.0.0.0:80

> This option configures `systemd` to create a listening TCP socket bound to all local IPv4 addresses on port 80.

WantedBy=sockets.target

> This is a special target used by all socket activated services. `systemd` will automatically bind to all such socket activation ports during boot up.

You also need to modify the `systemd.service` file as follows:

`/etc/systemd/system/www.example.com.service`

```
[Unit]
Description=Example Web Server

[Service]
ExecStart=/usr/bin/twistd \
    --nodaemon \
    --pidfile= \
    web --port systemd:domain=INET:index=0 --path .

NonBlocking=true

WorkingDirectory=/srv/www/www.example.com/static

User=nobody
Group=nobody

Restart=always
```

Note the following important directives and changes:

ExecStart

The `domain=INET` endpoint argument makes `twistd` treat the inherited file descriptor as an IPv4 socket.

The `index=0` endpoint argument makes `twistd` adopt the first file descriptor inherited from `systemd`.

Socket activation is also technically possible with other socket families and types, but Twisted currently only accepts IPv4 and IPv6 TCP sockets. See *Limitations and Known Issues* below.

NonBlocking

This must be set to `true` to ensure that `systemd` passes non-blocking sockets to Twisted.

[Install]

In this example, the `[Install]` section has been moved to the socket configuration file.

Reload `systemd` so that it reads the updated configuration files.

```
$ sudo systemctl daemon-reload
```

Start and enable the socket

You can now start `systemd` listening on the socket with the following command:

```
$ sudo systemctl start www.example.com.socket
```

This command refers specifically to the socket configuration file, **not** the service file.

`systemd` should now be listening on port 80

```
$ systemctl status www.example.com.socket
www.example.com.socket
          Loaded: loaded (/etc/systemd/system/www.example.com.socket; disabled)
          Active: active (listening) since Tue 2013-01-29 14:53:17 GMT; 7s ago

Jan 29 14:53:17 zorin.lan systemd[1]: Listening on www.example.com.socket.
```

But `twistd` should not yet have started. You can verify this using the `systemctl` command. eg

```
$ systemctl status www.example.com.service
www.example.com.service - Example Web Server
          Loaded: loaded (/etc/systemd/system/www.example.com.service; static)
          Active: inactive (dead) since Tue 2013-01-29 14:48:42 GMT; 6min ago
```

Enable the socket, so that it will be started automatically with the other socket activated services during boot up.

```
$ sudo systemctl enable www.example.com.socket
ln -s '/etc/systemd/system/www.example.com.socket' '/etc/systemd/system/sockets.
↪target.wants/www.example.com.socket'
```

Activate the port to start the service

Now try connecting to http://localhost:80 in your web browser.

`systemd` will accept the connection and start `twistd`, passing it the listening socket. You can verify this by using systemctl to report the status of the service. eg

```
$ systemctl status www.example.com.service
www.example.com.service - Example Web Server
          Loaded: loaded (/etc/systemd/system/www.example.com.service; static)
          Active: active (running) since Tue 2013-01-29 15:02:20 GMT; 3s ago
        Main PID: 25605 (twistd)
          CGroup: name=systemd:/system/www.example.com.service
                  -25605 /usr/bin/python /usr/bin/twistd --nodaemon --pidfile= web --
↪port systemd:domain=INET:index=0 --path .

Jan 29 15:02:20 zorin.lan systemd[1]: Started Example Web Server.
Jan 29 15:02:20 zorin.lan twistd[25605]: 2013-01-29 15:02:20+0000 [-] Log opened.
Jan 29 15:02:20 zorin.lan twistd[25605]: 2013-01-29 15:02:20+0000 [-] twistd 12.1.0 (/
↪usr/bin/python 2.7.3) starting up.
Jan 29 15:02:20 zorin.lan twistd[25605]: 2013-01-29 15:02:20+0000 [-] reactor class:␣
↪twisted.internet.epollreactor.EPollReactor.
Jan 29 15:02:20 zorin.lan twistd[25605]: 2013-01-29 15:02:20+0000 [-] Site starting␣
↪on 80
Jan 29 15:02:20 zorin.lan twistd[25605]: 2013-01-29 15:02:20+0000 [-] Starting␣
↪factory <twisted.web.server.Site instance at 0x24be758>
```

Conclusion

In this tutorial you have learned how to deploy a Twisted service using `systemd`. You have also learned how the service can be started on demand, using socket activation.

Limitations and Known Issues

1. Twisted can not accept datagram sockets from `systemd`.
2. Twisted does not support listening for SSL connections on sockets inherited from `systemd`.

Further Reading

- systemd Documentation

Logging with twisted.logger

The Basics

Logging consists of two main endpoints: applications that emit events, and observers that receive and handle those events. An event is simply a `dict` object containing the relevant data that describes something interesting that has occurred in the application. For example: a web server might emit an event after handling each request that includes the URI of requested resource, the response's status code, a count of bytes transferred, and so on. All of that information might be contained in a pair of objects representing the request and response, so logging this event could be as simple as:

```
log.info(request=request, response=response)
```

The above API would seem confusing to users of many logging systems, which are built around the idea of emitting strings to a file. There is, after all, no string in the above call. In such systems, one might expect the API to look like this instead:

```
log.info(
    "{uri}: status={status}, bytes={size}, etc..."
    .format(uri=request.uri, status=response.code, size=response.size)
)
```

Here, a string is rendered by formatting data from our request and response objects. The string can be easily appended to a log file, to be later read by an administrator, or perhaps teased out via some scripts or log gathering tools.

One disadvantage to this is that a strings meant to be human-readable are not necessarily easy (or even possible) for software to handle reliably. While text files are a common medium for storing logs, one might want to write code that notices certain types of events and does something other than store the event.

In the web server example, if many requests from a given IP address are resulting in failed authentication, someone might be trying to break in to the server. Perhaps blocking requests from that IP address would be useful. An observer receiving the above event as a string would have to resort to parsing the string to extract the relevant information, which may perform poorly and, depending on how well events are formatted, it may also be difficult to avoid bugs. Additionally, any information not encoded into the string is simply unavailable; if the IP address isn't in the string, it cannot be obtained from the event.

However, if the `request` and `response` objects are available to the observer, as in the first example, it would be possible to write an observer that accessed any attributes of those objects. It would also be a lot easier to write an observer that emitted structured information by serializing these objects into a format such as JSON, rows in a database, etc.

Events-as-strings do have the advantage that it's obvious what an observer that writes strings, for example to a file, would emit. We can solve this more flexibly by providing an optional format string in events that can be used for this purpose:

```
log.info(
    "{request.uri}: status={response.status}, bytes={response.size}, etc...",
    request=request, response=response
)
```

Note that this looks very much like the events-as-strings version of the API, except that the string is not rendered by the caller; we leave that to the observer, *if and when it is needed* . The observer also has direct access to the request and response objects, as well as their attributes. Now a text-based observer can format the text in a prescribed way, and an observer that wants to handle these events in some other manner can do so as well.

Usage for emitting applications

The first thing that an application that emits logging events needs to do is to instantiate a Logger object, which provides the API to emit events. A Logger may be created globally for a module:

```
from twisted.logger import Logger
log = Logger()

def handleData(data):
    log.debug("Got data: {data!r}.", data=data)
```

A Logger can also be associated with a class:

```
from twisted.logger import Logger

class Foo(object):
    log = Logger()

    def oops(self, data):
```

```
        self.log.error(
            "Oops! Invalid data from server: {data!r}",
            data=data
        )
```

When associated with a class in this manner, the `"log_source"` key is set in the event. For example:

`logsource.py`

```
from twisted.logger import Logger

class MyObject(object):
    log = Logger()

    def __init__(self, value):
        self.value = value

    def doSomething(self, something):
        self.log.info(
            "Object with value {log_source.value!r} doing {something}.",
            something=something
        )

MyObject(7).doSomething("a task")
```

This example will show the string "object with value 7 doing a task" because the `log_source` key is automatically set to the `MyObject` instance that the `Logger` is retrieved from.

Capturing Failures

Logger provides a failure method, which allows one to capture a Failure object conveniently:

```
from twisted.logger import Logger
log = Logger()

try:
    1 / 0
except:
    log.failure("Math is hard!")
```

The emitted event will have the `"log_failure"` key set, which is a Failure that captures the exception. This can be used by my observers to obtain a traceback. For example, FileLogObserver will append the traceback to it's output:

```
Math is hard!

Traceback (most recent call last):
--- <exception caught here> ---
  File "/tmp/test.py", line 8, in <module>
    1/0
exceptions.ZeroDivisionError: integer division or modulo by zero
```

Note that this API is meant to capture unexpected and unhandled errors (that is: bugs, which is why tracebacks are preserved). As such, it defaults to logging at the critical level. It is generally more appropriate to instead use *log.error()* when logging an expected error condition that was appropriately handled by the software.

Namespaces

All Logger s have a namespace, which can be used to categorize events. Namespaces may be specified by passing in a `namespace` argument to Logger 's initializer, but if none is given, the logger will derive its namespace from the module name of the callable that instantiated it, or, in the case of a class, from the fully qualified name of the class. A Logger will add a `log_namespace` key to the events it emits.

In the first example above, the namespace would be `some.module` , and in the second example, it would be `some.module.Foo` .

Log levels

Logger s provide a number of methods for emitting events. These methods all have the same signature, but each will attach a specific `log_level` key to events. Log levels are defined by the LogLevel constants container. These are:

debug

> Debugging events: Information of use to a developer of the software, not generally of interest to someone running the software unless they are attempting to diagnose a software issue.

info

> Informational events: Routine information about the status of an application, such as incoming connections, startup of a subsystem, etc.

warn

> Warning events: Events that may require greater attention than informational events but are not a systemic failure condition, such as authorization failures, bad data from a network client, etc. Such events are of potential interest to system administrators, and should ideally be phrased in such a way, or documented, so as to indicate an action that an administrator might take to mitigate the warning.

error

> Error conditions: Events indicating a systemic failure. For example, resource exhaustion, or the loss of connectivity to an external system, such as a database or API endpoint, without which no useful work can proceed. Similar to warnings, errors related to operational parameters may be actionable to system administrators and should provide references to resources which an administrator might use to resolve them.

critical

> Critical failures: Errors indicating systemic failure (ie. service outage), data corruption, imminent data loss, etc. which must be handled immediately. This includes errors unanticipated by the software, such as unhandled exceptions, wherein the cause and consequences are unknown.

In the first example above, the call to `log.debug` will add a `log_level` key to the emitted event with a value of LogLevel.debug . In the second example, calling `self.log.error` would use a value of LogLevel.error .

The above descriptions are simply guidance, but it is worth noting that log levels have a reduced value if they are used inconsistently. If one module in an application considers a message informational, and another module considers a similar message an error, then filtering based on log levels becomes harder. This is increasingly likely if the modules in question are developed by different parties, as will often be the case with externally source libraries and frameworks. (If a module tends to use higher levels than another, namespaces may be used to calibrate the relative use of log levels, but that is obviously suboptimal.) Sticking to the above guidelines will hopefully help here.

Emitter method signatures

The emitter methods (debug , info , warn , etc.) all take an optional format string as a first argument, followed by keyword arguments that will be included in the emitted event.

Note that all three examples in the opening section of this HOWTO fit this signature. The first omits the format, which doesn't lend itself well to text logging. The second omits the keyword arguments, which hostile to anything other than text logging, and is therefore ill-advised. Finally, the third provides both, which is the recommended usage.

These methods are all convenience wrappers around the emit method, which takes a LogLevel as its first argument.

Format strings

Format strings provide observers with a standard way to format an event as text suitable for a human being to read, via the function formatEvent . When writing a format string, take care to present it in a manner which would make as much sense as possible to a human reader. Particularly, format strings need not be written with an eye towards parseability or machine-readability. If you want to save your log events along with their structure and then analyze them later, see the next section, on *"saving events for later"* .

Format strings should be

`unicode` , and use PEP 3101 syntax to describe how the event should be rendered as human-readable text. For legacy support and convenience in python 2, UTF-8-encoded `bytes` are also accepted for format strings, but unicode is preferred. There are two variations from PEP 3101 in the format strings used by this module:

1. Positional (numerical) field names (eg. `{0}`) are not permitted. Event keys are not ordered, which means positional field names do not make sense in this context. However, this is not an accidental limitation, but an intentional design decision. As software evolves, log messages often grow to include additional information, while still logging the same conceptual event. By using meaningful names rather than opaque indexes for event keys, these identifiers are more robust against future changes in the format of messages and the information provided.

2. Field names ending in parentheses (eg. `{foo()}`) will call the referenced object with no arguments, then call `str` on the result, rather than calling `str` on the referenced object directly. This extension to PEP 3101 format syntax is provided to make it as easy as possible to defer potentially expensive work until a log message must be emitted. For example, let's say that we wanted to log a message with some useful, but potentially expensive information from the 'request' object:

   ```
   log.info("{request.uri} useful, but expensive: {request.usefulButExpensive()}",
            request=request)
   ```

   In the case where this log message is filtered out as uninteresting and not saved, no formatting work is done *at all* ; and since we can use PEP3101 attribute-access syntax in conjunction with this parenthesis extension, the caller does not even need to build a function or bound method object to pass as a separate key. There is no support for specifying arguments in the format string; the goal is to make it idiomatic to express that work be done later, not to implement a full Python expression evaluator.

Event keys added by the system

The logging system will add keys to emitted events. All event keys that are inserted by the logging system with have a `log_` prefix, to avoid namespace collisions with application-provided event keys. Applications should therefore not insert event keys using the `log_` prefix, as that prefix is reserved for the logging system. System-provided event keys include:

`log_logger`

Logger object that the event was emitted to.

`log_source`

The source object that emitted the event. When a Logger is accessed as an attribute of a class, the class is the source. When accessed as an attribute of an instance, the instance is the source. In other cases, the source is `None` .

`log_level`

The LogLevel associated with the event.

`log_namespace`

The namespace associated with the event.

`log_format`

The format string provided for use by observers that wish to render the event as text. This may be `None` , if no format string was provided.

`log_time`

The time that the event was emitted, as returned by time .

`log_failure`

A Failure object captured when the event was emitted.

Avoid mutable event keys

Emitting applications should be cautious about inserting objects into event which may be mutated later. While observers are called synchronously, it is possible that an observer will do something like queue up the event for later serialization, in which case the serialized object may be different than intended.

Saving events for later

For compatibility reasons, `twistd` will log to a text-based format by default. However, it's much better to use a structured log file format which preserves information about the events being logged. `twisted.logger` provides two APIs: jsonFileLogObserver and eventsFromJSONLogFile, which allow you to save and retrieve structured log events with a basic level of fidelity. Log events are serialized as JSON dictionaries, with serialization rules that are as lenient as possible; any unknown values are replaced with simple placeholder values.

`jsonFileLogObserver` will create a log observer that will save events as structured data, like so:

`saver.py`

```
import io
from twisted.logger import jsonFileLogObserver, Logger

log = Logger(observer=jsonFileLogObserver(io.open("log.json", "a")),
             namespace="saver")

def loggit(values):
    log.info("Some values: {values!r}", values=values)

loggit([1234, 5678])
loggit([9876, 5432])
```

And `eventsFromJSONLogFile` can load those events again; here, you can see that the event has preserved enough information to be formatted as human-readable again:

`loader.py`

```
import sys
import io
from twisted.logger import (
    eventsFromJSONLogFile, textFileLogObserver
)

output = textFileLogObserver(sys.stdout)

for event in eventsFromJSONLogFile(io.open("log.json")):
    output(event)
```

You can also, of course, feel free to access any of the keys in the `event` object as you load them, as well; basic structures such as lists, numbers, dictionaries and strings (anything serializable with JSON) will be preserved:

`loader-math.py`

```
from __future__ import print_function

import io
from twisted.logger import eventsFromJSONLogFile

for event in eventsFromJSONLogFile(io.open("log.json")):
    print(sum(event["values"]))
```

Implementing an observer

An observer must provide the ILogObserver interface. That interface simply describes a 1-argument callable that takes a `dict` , so a simple implementation may simply use the handy provider decorator on a function that takes one argument:

```
from zope.interface import provider
from twisted.logger import ILogObserver, formatEvent

@provider(ILogObserver)
def simpleObserver(event):
    print(formatEvent(event))
```

The formatEvent function returns a textual (`unicode`) representation of the event.

While it is recommended, in most cases it is not required that observers declare their compliance with ILogObserver . This flexibility exists to allow for pre-existing callables and lambda expressions to be used as observers. As an example, if one would like to accumulate events in a `list` , then `list.append` may be used as an observer.

When implementing your own log observer, however, you should always keep in mind that unlike most objects within Twisted, a log observer *must be thread safe* .

Specifically, a log observer:

- must be prepared to be called from threads other than the main thread (or I/O thread, or reactor thread)
- must be prepared to be called from multiple threads concurrently
- must not interact with other Twisted APIs that are not explicitly thread-safe without first taking precautions like using callFromThread

Keep in mind that this is true even if you elect not to explicitly interact with any threads from your program. Twisted itself may log messages from threads, and Twisted may internally use APIs like callInThread ; for example, Twisted uses threads to look up hostnames when making an outgoing connection.

Given this extra wrinkle, it's usually best to see if you can find an existing log observer implementation that does what you need before implementing your own; thread safety can be tricky to implement. Luckily, twisted.logger comes with several useful observers, which are documented below.

Writing an observer for event analysis

Twisted includes log observers which take care of most of the "normal" uses of logging, like writing messages to a file, saving them as text, filtering them, and so on. There are two common reasons you might want to write a log observer of your own. The first is to ship log messages over to a different kind of external system which Twisted itself does not support. If you've read the above, you now know enough to do that; simply implement a function that converts the dictionary to something amenable to your external system or file format, and send it there or save it. The second task is to do some kind of analysis, either real-time within your process or after the fact offline.

You're probably going to have to aggregate information from your own code and from libraries you're using, including Twisted itself, and you may not have much in the way of control over how those messages are organized. You're also probably trying to aggregate information from ad-hoc messages into some kind of structure.

One way to extract such semi-structured information is to feed your logs into an external system like logstash , and process them there. Such systems are quite powerful and `twisted.logger` does not try to replace them. However, such systems are necessarily external, and therefore if you want to react to the log analysis *within* your application - for example, denying access in response to an abusive client - you would have to write some glue code to push messages from your log-analysis system back into your application. For such cases, it's useful to be able to write log analysis code as a log observer.

Assuming that the libraries whose log events you're interested in analyzing are making use of `twisted.logger` , you can analyze log events either live as they're being logged, or loaded from a saved log file.

If you're writing code to log events directly for potential analysis, then you should simply structure your messages to include all necessary information as serialization-friendly values in events, and then simply pull them out, like the `loader-math.py` example above.

However, let's say you're trying to interact with a system that logs messages like so:

`ad_hoc.py`

```
from twisted.logger import Logger

class AdHoc(object):
    log = Logger(namespace="ad_hoc")

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def logMessage(self):
        self.log.info("message from {log_source} "
                      "where a is {log_source.a} and b is {log_source.b}")
```

In this example, the `AdHoc` object is not itself serializable, but it has two relevant attributes, which the log message's format string calls out as attributes of the `log_source` field, as described above: `a` and `b` .

To analyze this event, we can't pursue the same strategy shown above in `loader-math.py` . We don't have any key/value pairs of the log event to examine directly; `a` and `b` are not present as keys themselves. We could look for the

`log_source` key within the event and access its `a` and `b` attributes, but that wouldn't work once the event had been serialized and loaded again, since an `AdHoc` instance isn't a basic type that can be saved to JSON.

Luckily, twisted.logger provides an API for doing just this: extractField . You use it like so:

`analyze.py`

```
from __future__ import print_function

from twisted.logger import extractField

fmt = (
    "message from {log_source} "
    "where a is {log_source.a} and b is {log_source.b}"
)

def analyze(event):
    if event.get("log_format") == fmt:
        a = extractField("log_source.a", event)
        b = extractField("log_source.b", event)
        print("A + B = " + repr(a + b))
```

This `analyze` function can then be used in two ways. First, you can analyze your application "live", as it's running.

`online_analyze.py`

```
from twisted.logger import globalLogPublisher
from analyze import analyze
from ad_hoc import AdHoc

globalLogPublisher.addObserver(analyze)

AdHoc(3, 4).logMessage()
```

If you run this script, you will see that it extracts the values from the `AdHoc` object's log message.

However, you can also analyze output from a saved log file, using the exact same code. First, you'll need to produce a log file with a message from an AdHoc object in it:

`ad_hoc_save.py`

```
import io
from twisted.logger import jsonFileLogObserver, globalLogPublisher
from ad_hoc import AdHoc

globalLogPublisher.addObserver(jsonFileLogObserver(io.open("log.json", "a")))

AdHoc(3, 4).logMessage()
```

If you run that script, it will produce a `log.json` file which can be analyzed in the same manner as the `loader.py` example above:

`offline_analyze.py`

```
import io
from twisted.logger import eventsFromJSONLogFile
from analyze import analyze

for event in eventsFromJSONLogFile(io.open("log.json")):
    analyze(event)
```

When doing analysis, it always seems like you should have saved more structured data for later. However, log messages are often written early on in development, in an ad-hoc way, before a solid understanding has developed about what information will be useful and what will be redundant. Log messages are therefore a stream-of-consciousness commentary on what is going on as a system is being developed.

`extractField` lets you acknowledge the messy reality of how log messages are written, but still take advantage of structured analysis later on. Just always be sure to use event format fields, not string concatenation, to reference information about a particular event, and you'll be able to easily pull apart hastily-written ad-hoc messages from multiple versions of a system, either as it's running or once you've saved a log file.

Registering an observer

One way to register an observer is to construct a Logger object with it:

```
from twisted.logger import Logger
from myobservers import PrintingObserver

log = Logger(observer=PrintingObserver())

log.info("Hello")
```

This will cause all of a logger's events to be sent to the given observer. In the above example, all events emitted by the logger (eg. `"Hello"`) will be printed. While that is useful in some cases, it is common to register multiple observers, and to do so globally for all (or most) loggers in an application.

The global log publisher

The global log publisher is a log observer whose purpose is to capture log events not directed to a specific observer. In a typical application, the majority of log events will be emitted to the global log publisher. Observers can register themselves with the global log publisher in order to be forwarded these events.

When a Logger is created without specifying an observer to send events to, the logger will send its events to the global log publisher, which is accessible via the name globalLogPublisher .

The global log publisher is a singleton instance of a private subclass of LogPublisher , which is itself an ILogObserver . What this means is that the global log publisher accepts events like any other observer, and that it forwards those events to other observers. Observers can be registered to be forwarded events by calling the LogPublisher method addObserver , and unregister by calling removeObserver :

```
from twisted.logger import globalLogPublisher
from myobservers import PrintingObserver

log = Logger()

globalLogPublisher.addObserver(PrintingObserver())

log.info("Hello")
```

The result here is the same as the previous example, except that additional observers can be (and may already have been) registered. We know that `"Hello"` will be printed. We don't know, but it's very possible, that the same event will also be handled by other observers.

There is no supported API to discover what other observers are registered with a LogPublisher ; in general, one doesn't need to know. If an application is running in `twistd` , for example, it's likely that an observer is streaming events to a file by the time the application code is in play. If it is running in a `twistd` web container, there will probably be another observer writing to the access log.

A caveat here is that events are `dict` objects, which are mutable, so it is possible for an observer to modify an event that it sees. Because doing so will modify what other observers will see, modifying a received event can be problematic and should be strongly discouraged. It can be particularly harmful to edit or remove the content of an existing event key. Furthermore, no guarantees are made as to the order in which observers are called, so the effect of such modifications on other observers may be non-deterministic.

Starting the global log publisher

When the global log publisher is created, it uses a LimitedHistoryLogObserver (see below) to store events that are logged by the application in memory until logging is started. Logging is started by registering the first set of observers with the global log publisher by calling beginLoggingTo :

```
from twisted.logger import globalLogBeginner
from myobservers import PrintingObserver

log = Logger()

log.info("Hello")

observers = [PrintingObserver()]

globalLogBeginner.beginLoggingTo(observers)

log.info("Hello, again")
```

This:

- Adds the given observers (in this example, the `PrintingObserver`) to the global log observer
- Forwards all of the events that were stored in memory prior to calling beginLoggingTo to these observers
- Gets rid of the LimitedHistoryLogObserver , as it is no longer needed.

It is an error to call beginLoggingTo more than once.

Note: If the global log publisher is never started, the in-memory event buffer holds (a bounded number of) log events indefinitely. This may unexpectedly increase application memory or CPU usage. It is highly recommended that the global log publisher be started as early as feasible.

Provided log observers

This module provides a number of pre-built observers for applications to use:

LogPublisher

> Forwards events to other publishers. This allows one to create a graph of observers.

LimitedHistoryLogObserver

> Stores a limited number of received events, and can re-play those stored events to another observer later. This is useful for keeping recent logging history in memory for inspection when other log outputs are not available.

FileLogObserver

> Formats events as text, prefixed with a time stamp and a "system identifier", and writes them to a file. The system identifier defaults to a combination of the event's namespace and level.

FilteringLogObserver

Forwards events to another observer after applying a set of filter predicates (providers of ILogFilterPredicate). LogLevelFilterPredicate is a predicate that be configured to keep track of which log levels to filter for different namespaces, and will filter out events that are not at the appropriate level or higher.

Compatibility with standard library logging

STDLibLogObserver is provided for compatibility with the standard library's logging module. Log levels are mapped between the two systems, and the various attributes of standard library log records are filled in properly.

Note that standard library logging is a blocking API, and logging can be configured to block for long periods (eg. it may write to the network). No protection is provided to prevent blocking, so such configurations may cause Twisted applications to perform poorly.

Compatibility with twisted.python.log

This module provides some facilities to enable the existing twisted.python.log module to compatibly forward it's messages to this module. As such, existing clients of twisted.python.log will begin using this module indirectly, with no changes to the older module's API.

Incrementally porting observers

Observers have an incremental path for porting to the new module. LegacyLogObserverWrapper is an ILogObserver that wraps a log observer written for the older module. This allows an old-style observer to be registered with a new-style logger or log publisher compatibly.

Twisted's Legacy Logging System: `twisted.python.log`

Note: There is now a new logging system in Twisted (*you can read about how to use it here* and its API reference here) which is a replacement for twisted.python.log.

The old logging API, described here, remains for compatibility, and is now implemented as a client of the new logging system.

New code should adopt the new API.

Basic usage

Twisted provides a simple and flexible logging system in the twisted.python.log module. It has three commonly used functions:

msg

Logs a new message. For example:

```
from twisted.python import log
log.msg('Hello, world.')
```

err

Writes a failure to the log, including traceback information (if any). You can pass it a Failure or Exception instance, or nothing. If you pass something else, it will be converted to a string with `repr` and logged.

If you pass nothing, it will construct a Failure from the currently active exception, which makes it convenient to use in an `except` clause:

```
try:
    x = 1 / 0
except:
    log.err()    # will log the ZeroDivisionError
```

startLogging

Starts logging to a given file-like object. For example:

```
log.startLogging(open('/var/log/foo.log', 'w'))
```

or:

```
log.startLogging(sys.stdout)
```

or:

```
from twisted.python.logfile import DailyLogFile

log.startLogging(DailyLogFile.fromFullPath("/var/log/foo.log"))
```

By default, `startLogging` will also redirect anything written to `sys.stdout` and `sys.stderr` to the log. You can disable this by passing `setStdout=False` to `startLogging` .

Before `startLogging` is called, log messages will be discarded and errors will be written to stderr.

Logging and twistd

If you are using `twistd` to run your daemon, it will take care of calling `startLogging` for you, and will also rotate log files. See *twistd and tac* and the `twistd` man page for details of using twistd.

Log files

The twisted.python.logfile module provides some standard classes suitable for use with `startLogging` , such as DailyLogFile , which will rotate the log to a new file once per day.

Using the standard library logging module

If your application uses the Python standard library logging module or you want to use its easy configuration but don't want to lose twisted-produced messages, the observer PythonLoggingObserver should be useful to you.

You just start it like any other observer:

```
observer = log.PythonLoggingObserver()
observer.start()
```

Then configure the standard library logging module to behave as you want.

This method allows you to customize the log level received by the standard library logging module using the `logLevel` keyword:

```
log.msg("This is important!", logLevel=logging.CRITICAL)
log.msg("Don't mind", logLevel=logging.DEBUG)
```

Unless `logLevel` is provided, logging.INFO is used for `log.msg` and `logging.ERROR` is used for `log.err` .

One special care should be made when you use special configuration of the standard library logging module: some handlers (e.g. SMTP, HTTP) use the network and so can block inside the reactor loop. *Nothing* in `PythonLoggingObserver` is done to prevent that.

Writing log observers

Log observers are the basis of the Twisted logging system. Whenever `log.msg` (or `log.err`) is called, an event is emitted. The event is passed to each observer which has been registered. There can be any number of observers, and each can treat the event in any way desired. An example of a log observer in Twisted is the `emit` method of FileLogObserver . `FileLogObserver` , used by `startLogging` , writes events to a log file. A log observer is just a callable that accepts a dictionary as its only argument. You can then register it to receive all log events (in addition to any other observers):

```
twisted.python.log.addObserver(yourCallable)
```

The dictionary will have at least two items:

message

> The message (a list, usually of strings) for this log event, as passed to `log.msg` or the message in the failure passed to `log.err` .

isError

> This is a boolean that will be true if this event came from a call to `log.err` . If this is set, there may be a `failure` item in the dictionary as will, with a Failure object in it.

Other items the built in logging functionality may add include:

printed

> This message was captured from `sys.stdout` , i.e. this message came from a `print` statement. If `isError` is also true, it came from `sys.stderr` .

You can pass additional items to the event dictionary by passing keyword arguments to `log.msg` and `log.err` . The standard log observers will ignore dictionary items they don't use.

Important notes:

- Never block in a log observer, as it may run in main Twisted thread. This means you can't use socket or syslog standard library logging backends.
- The observer needs to be thread safe if you anticipate using threads in your program.

Customizing `twistd` logging

The behavior of the logging that `twistd` does can be customized either with the `--logger` option or by setting the `ILogObserver` component on the application object. See the *Application document* for more information.

Symbolic Constants

Note: `twisted.python.constants` has been deprecated in Twisted, and the same code has been spun out into Constantly. The API is identical, just install it from PyPI and replace `import twisted.python.constants` with `import constantly` in your code.

Overview

It is often useful to define names which will be treated as constants. twisted.python.constants provides APIs for defining such symbolic constants with minimal overhead and some useful features beyond those afforded by the common Python idioms for this task.

This document will explain how to use these APIs and what circumstances they might be helpful in.

Constant Names

Constants which have no value apart from their name and identity can be defined by subclassing Names . Consider this example, in which some HTTP request method constants are defined.

```
from twisted.python.constants import NamedConstant, Names
class METHOD(Names):
    """
    Constants representing various HTTP request methods.
    """
    GET = NamedConstant()
    PUT = NamedConstant()
    POST = NamedConstant()
    DELETE = NamedConstant()
```

Only direct subclasses of `Names` are supported (i.e., you cannot subclass `METHOD` to add new constants the collection).

Given this definition, constants can be looked up by name using attribute access on the `METHOD` object:

```
>>> METHOD.GET
<METHOD=GET>
>>> METHOD.PUT
<METHOD=PUT>
>>>
```

If it's necessary to look up constants from a string (e.g. based on user input of some sort), a safe way to do it is using `lookupByName` :

```
>>> METHOD.lookupByName('GET')
<METHOD=GET>
>>> METHOD.lookupByName('__doc__')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "twisted/python/constants.py", line 145, in lookupByName
    raise ValueError(name)
ValueError: __doc__
>>>
```

As demonstrated, it is safe because any name not associated with a constant (even those special names initialized by Python itself) will result in `ValueError` being raised, not some other object not intended to be used the way the constants are used.

The constants can also be enumerated using the `iterconstants` method:

```
>>> list(METHOD.iterconstants())
[<METHOD=GET>, <METHOD=PUT>, <METHOD=POST>, <METHOD=DELETE>]
>>>
```

Constants can be compared for equality or identity:

```
>>> METHOD.GET is METHOD.GET
True
>>> METHOD.GET == METHOD.GET
True
>>> METHOD.GET is METHOD.PUT
False
>>> METHOD.GET == METHOD.PUT
False
>>>
```

Ordered comparisons (and therefore sorting) also work. The order is defined to be the same as the instantiation order of the constants:

```
>>> from twisted.python.constants import NamedConstant, Names
>>> class Letters(Names):
...     a = NamedConstant()
...     b = NamedConstant()
...     c = NamedConstant()
...
>>> Letters.a < Letters.b < Letters.c
True
>>> Letters.a > Letters.b
False
>>> sorted([Letters.b, Letters.a, Letters.c])
[<Letters=a>, <Letters=b>, <Letters=c>]
>>>
```

A subclass of `Names` may define class methods to implement custom functionality. Consider this definition of `METHOD`:

```
from twisted.python.constants import NamedConstant, Names
class METHOD(Names):
    """
    Constants representing various HTTP request methods.
    """
    GET = NamedConstant()
    PUT = NamedConstant()
    POST = NamedConstant()
    DELETE = NamedConstant()

    @classmethod
    def isIdempotent(cls, method):
        """
        Return True if the given method is side-effect free, False otherwise.
        """
        return method is cls.GET
```

This functionality can be used as any class methods are used:

```
>>> METHOD.isIdempotent(METHOD.GET)
True
>>> METHOD.isIdempotent(METHOD.POST)
```

```
False
>>>
```

Constants With Values

Constants with a particular associated value are supported by the Values base class. Consider this example, in which some HTTP status code constants are defined.

```
from twisted.python.constants import ValueConstant, Values
class STATUS(Values):
    """
    Constants representing various HTTP status codes.
    """
    OK = ValueConstant("200")
    FOUND = ValueConstant("302")
    NOT_FOUND = ValueConstant("404")
```

As with `Names` , constants are accessed as attributes of the class object:

```
>>> STATUS.OK
<STATUS=OK>
>>> STATUS.FOUND
<STATUS=FOUND>
>>>
```

Additionally, the values of the constants can be accessed using the `value` attribute of one these objects:

```
>>> STATUS.OK.value
'200'
>>>
```

As with `Names` , constants can be looked up by name:

```
>>> STATUS.lookupByName('NOT_FOUND')
<STATUS=NOT_FOUND>
>>>
```

Constants on a `Values` subclass can also be looked up by value:

```
>>> STATUS.lookupByValue('404')
<STATUS=NOT_FOUND>
>>> STATUS.lookupByValue('500')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "twisted/python/constants.py", line 244, in lookupByValue
      raise ValueError(value)
ValueError: 500
>>>
```

Multiple constants may have the same value. If they do, `lookupByValue` will find the one which is defined first.

Iteration is also supported:

```
>>> list(STATUS.iterconstants())
[<STATUS=OK>, <STATUS=FOUND>, <STATUS=NOT_FOUND>]
>>>
```

Constants can be compared for equality, identity and ordering:

```
>>> STATUS.OK == STATUS.OK
True
>>> STATUS.OK is STATUS.OK
True
>>> STATUS.OK is STATUS.NOT_FOUND
False
>>> STATUS.OK == STATUS.NOT_FOUND
False
>>> STATUS.NOT_FOUND > STATUS.OK
True
>>> STATUS.FOUND < STATUS.OK
False
>>>
```

Note that like `Names` , `Values` are ordered by instantiation order, not by value, though either order is the same in the above example.

As with `Names` , a subclass of `Values` can define custom methods:

```
from twisted.python.constants import ValueConstant, Values
class STATUS(Values):
    """
    Constants representing various HTTP status codes.
    """
    OK = ValueConstant("200")
    NO_CONTENT = ValueConstant("204")
    NOT_MODIFIED = ValueConstant("304")
    NOT_FOUND = ValueConstant("404")

    @classmethod
    def hasBody(cls, status):
        """
        Return True if the given status is associated with a response body,
        False otherwise.
        """
        return status not in (cls.NO_CONTENT, cls.NOT_MODIFIED)
```

This functionality can be used as any class methods are used:

```
>>> STATUS.hasBody(STATUS.OK)
True
>>> STATUS.hasBody(STATUS.NO_CONTENT)
False
>>>
```

Constants As Flags

Integers are often used as a simple set for constants. The values for these constants are assigned as powers of two so that bits in the integer can be set to represent them. Individual bits are often called *flags* . Flags supports this use-case, including allowing constants with particular bits to be set, for interoperability with other tools.

POSIX filesystem access control is traditionally done using a bitvector defining which users and groups may perform which operations on a file. This state might be represented using `Flags` as follows:

```
from twisted.python.constants import FlagConstant, Flags
class Permission(Flags):
    """
    Constants representing user, group, and other access bits for reading,
    writing, and execution.
    """
    OTHER_EXECUTE = FlagConstant()
    OTHER_WRITE = FlagConstant()
    OTHER_READ = FlagConstant()
    GROUP_EXECUTE = FlagConstant()
    GROUP_WRITE = FlagConstant()
    GROUP_READ = FlagConstant()
    USER_EXECUTE = FlagConstant()
    USER_WRITE = FlagConstant()
    USER_READ = FlagConstant()
```

As for the previous types of constants, these can be accessed as attributes of the class object:

```
>>> Permission.USER_READ
<Permission=USER_READ>
>>> Permission.USER_WRITE
<Permission=USER_WRITE>
>>> Permission.USER_EXECUTE
<Permission=USER_EXECUTE>
>>>
```

These constant objects also have a `value` attribute giving their integer value:

```
>>> Permission.USER_READ.value
256
>>>
```

These constants can be looked up by name or value:

```
>>> Permission.lookupByName('USER_READ') is Permission.USER_READ
True
>>> Permission.lookupByValue(256) is Permission.USER_READ
True
>>>
```

Constants can also be combined using the logical operators `&` (*and*), `|` (*or*), and `^` (*exclusive or*).

```
>>> Permission.USER_READ | Permission.USER_WRITE
<Permission={USER_READ,USER_WRITE}>
>>> (Permission.USER_READ | Permission.USER_WRITE) & Permission.USER_WRITE
<Permission=USER_WRITE>
>>> (Permission.USER_READ | Permission.USER_WRITE) ^ Permission.USER_WRITE
<Permission=USER_READ>
>>>
```

These combined constants can be deconstructed via iteration:

```
>>> mode = Permission.USER_READ | Permission.USER_WRITE
>>> list(mode)
[<Permission=USER_READ>, <Permission=USER_WRITE>]
>>> Permission.USER_READ in mode
True
>>> Permission.USER_EXECUTE in mode
```

```
False
>>>
```

They can also be inspected via boolean operations:

```
>>> Permission.USER_READ & mode
<Permission=USER_READ>
>>> bool(Permission.USER_READ & mode)
True
>>> Permission.USER_EXECUTE & mode
<Permission={}>
>>> bool(Permission.USER_EXECUTE & mode)
False
>>>
```

The unary operator ~ (*not*) is also defined:

```
>>> ~Permission.USER_READ
<Permission={GROUP_EXECUTE,GROUP_READ,GROUP_WRITE,OTHER_EXECUTE,OTHER_READ,OTHER_
↪WRITE,USER_EXECUTE,USER_WRITE}>
>>>
```

Constants created using these operators also have a `value` attribute.

```
>>> (~Permission.USER_WRITE).value
383
>>>
```

Note the care taken to ensure the ~ operator is applied first and the `value` attribute is looked up second.

A `Flags` subclass can also define methods, just as a `Names` or `Values` subclass may. For example, `Permission` might benefit from a method to format a flag as a string in the traditional style. Consider this addition to that class:

```
from twisted.python import filepath
from twisted.python.constants import FlagConstant, Flags
class Permission(Flags):
    ...

    @classmethod
    def format(cls, permissions):
        """
        Format permissions flags in the traditional 'rwxr-xr-x' style.
        """
        return filepath.Permissions(permissions.value).shorthand()
```

Use this like any other class method:

```
>>> Permission.format(Permission.USER_READ | Permission.USER_WRITE | Permission.GROUP_
↪READ | Permission.OTHER_READ)
'rw-r--r--'
>>>
```

twisted.enterprise.adbapi: Twisted RDBMS support

Abstract

Twisted is an asynchronous networking framework, but most database API implementations unfortunately have blocking interfaces – for this reason, twisted.enterprise.adbapi was created. It is a non-blocking interface to the standardized DB-API 2.0 API, which allows you to access a number of different RDBMSes.

What you should already know

- Python :-)
- How to write a simple Twisted Server (see *this tutorial* to learn how)
- Familiarity with using database interfaces (see the documentation for DBAPI 2.0)

Quick Overview

Twisted is an asynchronous framework. This means standard database modules cannot be used directly, as they typically work something like:

```
# Create connection...
db = dbmodule.connect('mydb', 'andrew', 'password')
# ...which blocks for an unknown amount of time

# Create a cursor
cursor = db.cursor()

# Do a query...
resultset = cursor.query('SELECT * FROM table WHERE ...')
# ...which could take a long time, perhaps even minutes.
```

Those delays are unacceptable when using an asynchronous framework such as Twisted. For this reason, Twisted provides twisted.enterprise.adbapi, an asynchronous wrapper for any DB-API 2.0-compliant module.

adbapi will do blocking database operations in separate threads, which trigger callbacks in the originating thread when they complete. In the meantime, the original thread can continue doing normal work, like servicing other requests.

How do I use adbapi?

Rather than creating a database connection directly, use the adbapi.ConnectionPool class to manage a connections for you. This allows adbapi to use multiple connections, one per thread. This is easy:

```
# Using the "dbmodule" from the previous example, create a ConnectionPool
from twisted.enterprise import adbapi
dbpool = adbapi.ConnectionPool("dbmodule", 'mydb', 'andrew', 'password')
```

Things to note about doing this:

- There is no need to import dbmodule directly. You just pass the name to adbapi.ConnectionPool's constructor.
- The parameters you would pass to dbmodule.connect are passed as extra arguments to adbapi.ConnectionPool's constructor. Keyword parameters work as well.

Now we can do a database query:

```
# equivalent of cursor.execute(statement), return cursor.fetchall():
def getAge(user):
    return dbpool.runQuery("SELECT age FROM users WHERE name = ?", user)

def printResult(l):
    if l:
        print(l[0][0], "years old")
    else:
        print("No such user")

getAge("joe").addCallback(printResult)
```

This is straightforward, except perhaps for the return value of `getAge`. It returns a Deferred, which allows arbitrary callbacks to be called upon completion (or upon failure). More documentation on Deferred is available *here*.

In addition to `runQuery`, there is also `runOperation` and `runInteraction` that gets called with a callable (e.g. a function). The function will be called in the thread with a adbapi.Transaction, which basically mimics a DB-API cursor. In all cases a database transaction will be committed after your database usage is finished, unless an exception is raised in which case it will be rolled back.

```
def _getAge(txn, user):
    # this will run in a thread, we can use blocking calls
    txn.execute("SELECT * FROM foo")
    # ... other cursor commands called on txn ...
    txn.execute("SELECT age FROM users WHERE name = ?", user)
    result = txn.fetchall()
    if result:
        return result[0][0]
    else:
        return None

def getAge(user):
    return dbpool.runInteraction(_getAge, user)

def printResult(age):
    if age != None:
        print(age, "years old")
    else:
        print("No such user")

getAge("joe").addCallback(printResult)
```

Also worth noting is that these examples assumes that dbmodule uses the "qmarks" paramstyle (see the DB-API specification). If your dbmodule uses a different paramstyle (e.g. pyformat) then use that. Twisted doesn't attempt to offer any sort of magic parameter munging – `runQuery(query, params, ...)` maps directly onto `cursor.execute(query, params, ...)`.

Examples of various database adapters

Notice that the first argument is the module name you would usually import and get `connect(...)` from, and that following arguments are whatever arguments you'd call `connect(...)` with.

```
from twisted.enterprise import adbapi

# PostgreSQL PyPgSQL
cp = adbapi.ConnectionPool("pyPgSQL.PgSQL", database="test")
```

```
# MySQL
cp = adbapi.ConnectionPool("MySQLdb", db="test")
```

And that's it!

That's all you need to know to use a database from within Twisted. You probably should read the adbapi module's documentation to get an idea of the other functions it has, but hopefully this document presents the core ideas.

Parsing command-lines with usage.Options

Introduction

There is frequently a need for programs to parse a UNIX-like command line program: options preceded by `-` or `--`, sometimes followed by a parameter, followed by a list of arguments. The twisted.python.usage provides a class, `Options`, to facilitate such parsing.

While Python has the `getopt` module for doing this, it provides a very low level of abstraction for options. Twisted has a higher level of abstraction, in the class twisted.python.usage.Options. It uses Python's reflection facilities to provide an easy to use yet flexible interface to the command line. While most command line processors either force the application writer to write their own loops, or have arbitrary limitations on the command line (the most common one being not being able to have more than one instance of a specific option, thus rendering the idiom `program -v -v -v` impossible), Twisted allows the programmer to decide how much control they want.

The `Options` class is used by subclassing. Since a lot of time it will be used in the twisted.tap package, where the local conventions require the specific options parsing class to also be called `Options`, it is usually imported with

```
from twisted.python import usage
```

Boolean Options

For simple boolean options, define the attribute `optFlags` like this:

```
class Options(usage.Options):

    optFlags = [["fast", "f", "Act quickly"], ["safe", "s", "Act safely"]]
```

`optFlags` should be a list of 3-lists. The first element is the long name, and will be used on the command line as `--fast`. The second one is the short name, and will be used on the command line as `-f`. The last element is a description of the flag and will be used to generate the usage information text. The long name also determines the name of the key that will be set on the Options instance. Its value will be 1 if the option was seen, 0 otherwise. Here is an example for usage:

```
from __future__ import print_function

class Options(usage.Options):

    optFlags = [
        ["fast", "f", "Act quickly"],
        ["good", "g", "Act well"],
        ["cheap", "c", "Act cheaply"]
    ]
```

```
command_line = ["-g", "--fast"]

options = Options()
try:
    options.parseOptions(command_line)
except usage.UsageError as errortext:
    print('{}: {}'.format(sys.argv[0], errortext))
    print('{}: Try --help for usage details.'.format(sys.argv[0]))
    sys.exit(1)
if options['fast']:
    print("fast", end='')
if options['good']:
    print("good", end='')
if options['cheap']:
    print("cheap", end='')
print()
```

The above will print `fast good`.

Note here that Options fully supports the mapping interface. You can access it mostly just like you can access any other dict. Options are stored as mapping items in the Options instance: parameters as 'paramname': 'value' and flags as 'flagname': 1 or 0.

Inheritance, Or: How I Learned to Stop Worrying and Love the Superclass

Sometimes there is a need for several option processors with a unifying core. Perhaps you want all your commands to understand `-q`/`--quiet` means to be quiet, or something similar. On the face of it, this looks impossible: in Python, the subclass's `optFlags` would shadow the superclass's. However, `usage.Options` uses special reflection code to get all of the `optFlags` defined in the hierarchy. So the following:

```
class BaseOptions(usage.Options):

    optFlags = [["quiet", "q", None]]

class SpecificOptions(BaseOptions):

    optFlags = [
        ["fast", "f", None], ["good", "g", None], ["cheap", "c", None]
    ]
```

Is the same as:

```
class SpecificOptions(usage.Options):

    optFlags = [
        ["quiet", "q", "Silence output"],
        ["fast", "f", "Run quickly"],
        ["good", "g", "Don't validate input"],
        ["cheap", "c", "Use cheap resources"]
    ]
```

Parameters

Parameters are specified using the attribute `optParameters`. They *must* be given a default. If you want to make sure you got the parameter from the command line, give a non-string default. Since the command line only has strings,

this is completely reliable.

Here is an example:

```
from __future__ import print_function

from twisted.python import usage

class Options(usage.Options):

    optFlags = [
        ["fast", "f", "Run quickly"],
        ["good", "g", "Don't validate input"],
        ["cheap", "c", "Use cheap resources"]
    ]
    optParameters = [["user", "u", None, "The user name"]]

config = Options()
try:
    config.parseOptions() # When given no argument, parses sys.argv[1:]
except usage.UsageError as errortext:
    print('{}: {}'.format(sys.argv[0], errortext))
    print('{}: Try --help for usage details.'.format(sys.argv[0]))
    sys.exit(1)

if config['user'] is not None:
    print("Hello", config['user'])
print("So, you want it:")

if config['fast']:
    print("fast", end='')
if config['good']:
    print("good", end='')
if config['cheap']:
    print("cheap", end='')
print()
```

Like `optFlags` , `optParameters` works smoothly with inheritance.

Option Subcommands

It is useful, on occasion, to group a set of options together based on the logical "action" to which they belong. For this, the `usage.Options` class allows you to define a set of "subcommands" , each of which can provide its own `usage.Options` instance to handle its particular options.

Here is an example for an Options class that might parse options like those the cvs program takes

```
from twisted.python import usage

class ImportOptions(usage.Options):
    optParameters = [
        ['module', 'm', None, None], ['vendor', 'v', None, None],
        ['release', 'r', None]
    ]

class CheckoutOptions(usage.Options):
    optParameters = [['module', 'm', None, None], ['tag', 'r', None, None]]
```

```
class Options(usage.Options):
    subCommands = [['import', None, ImportOptions, "Do an Import"],
                   ['checkout', None, CheckoutOptions, "Do a Checkout"]]

    optParameters = [
        ['compression', 'z', 0, 'Use compression'],
        ['repository', 'r', None, 'Specify an alternate repository']
    ]

config = Options(); config.parseOptions()
if config.subCommand == 'import':
    doImport(config.subOptions)
elif config.subCommand == 'checkout':
    doCheckout(config.subOptions)
```

The `subCommands` attribute of `Options` directs the parser to the two other `Options` subclasses when the strings `"import"` or `"checkout"` are present on the command line. All options after the given command string are passed to the specified Options subclass for further parsing. Only one subcommand may be specified at a time. After parsing has completed, the Options instance has two new attributes - `subCommand` and `subOptions` - which hold the command string and the Options instance used to parse the remaining options.

Generic Code For Options

Sometimes, just setting an attribute on the basis of the options is not flexible enough. In those cases, Twisted does not even attempt to provide abstractions such as "counts" or "lists" , but rather lets you call your own method, which will be called whenever the option is encountered.

Here is an example of counting verbosity

```
from twisted.python import usage

class Options(usage.Options):

    def __init__(self):
        usage.Options.__init__(self)
        self['verbosity'] = 0 # default

    def opt_verbose(self):
        self['verbosity'] = self['verbosity']+1

    def opt_quiet(self):
        self['verbosity'] = self['verbosity']-1

    opt_v = opt_verbose
    opt_q = opt_quiet
```

Command lines that look like `command -v -v -v -v` will increase verbosity to 4, while `command -q -q -q` will decrease verbosity to -3.

The usage.Options class knows that these are parameter-less options, since the methods do not receive an argument. Here is an example for a method with a parameter:

```
from twisted.python import usage

class Options(usage.Options):

    def __init__(self):
```

```
        usage.Options.__init__(self)
        self['symbols'] = []

    def opt_define(self, symbol):
        self['symbols'].append(symbol)

    opt_D = opt_define
```

This example is useful for the common idiom of having `command -DFOO -DBAR` to define symbols.

Parsing Arguments

`usage.Options` does not stop helping when the last parameter is gone. All the other arguments are sent into a function which should deal with them. Here is an example for a `cmp` like command.

```
from twisted.python import usage

class Options(usage.Options):

    optParameters = [["max_differences", "d", 1, None]]

    def parseArgs(self, origin, changed):
        self['origin'] = origin
        self['changed'] = changed
```

The command should look like `command origin changed`.

If you want to have a variable number of left-over arguments, just use `def parseArgs(self, *args):`. This is useful for commands like the UNIX `cat(1)`.

Post Processing

Sometimes, you want to perform post processing of options to patch up inconsistencies, and the like. Here is an example:

```
from twisted.python import usage

class Options(usage.Options):

    optFlags = [
        ["fast", "f", "Run quickly"],
        ["good", "g", "Don't validate input"],
        ["cheap", "c", "Use cheap resources"]
    ]

    def postOptions(self):
        if self['fast'] and self['good'] and self['cheap']:
            raise usage.UsageError("can't have it all, brother")
```

Type enforcement

By default, all options are handled as strings. You may want to enforce the type of your option in some specific case, the classic example being port number. Any callable can be specified in the fifth row of `optParameters` and will be called with the string value passed in parameter.

```
from twisted.python import usage

class Options(usage.Options):
    optParameters = [
            ["shiny_integer", "s", 1, None, int],
            ["dummy_float", "d", 3.14159, None, float],
        ]
```

Note that default values are not coerced, so you should either declare it with the good type (as above) or handle it when you use your options.

The coerce function may have a coerceDoc attribute, the content of which will be printed after the documentation of the option. It's particularly useful for reusing the function at multiple places.

```
def oneTwoThree(val):
    val = int(val)
    if val not in range(1, 4):
        raise ValueError("Not in range")
    return val
oneTwoThree.coerceDoc = "Must be 1, 2 or 3."

from twisted.python import usage

class Options(usage.Options):
    optParameters = [["one_choice", "o", 1, None, oneTwoThree]]
```

This example code will print the following help when added to your program:

```
$ python myprogram.py --help
Usage: myprogram [options]
Options:
  -o, --one_choice=           [default: 0]. Must be 1, 2 or 3.
```

Shell tab-completion

The `Options` class may provide tab-completion to interactive command shells. Only `zsh` is supported at present, but there is some interest in supporting `bash` in the future.

Support is automatic for all of the commands shipped with Twisted. Zsh has shipped, for a number of years, a completion function which ties in to the support provided by the `Options` class.

If you are writing a `twistd` plugin, then tab-completion for your `twistd` sub-command is also automatic.

For other commands you may easily provide zsh tab-completion support. Copy the file "twisted/python/twisted-completion.zsh" and name it something like "_mycommand". A leading underscore with no extension is zsh's convention for completion function files.

Edit the new file and change the first line to refer only to your new command(s), like so:

```
#compdef mycommand
```

Then ensure this file is made available to the shell by placing it in one of the directories appearing in zsh's $fpath. Restart zsh, and ensure advanced completion is enabled (`autoload -U compinit; compinit`) . You should then be able to type the name of your command and press Tab to have your command-line options completed.

Completion metadata

Optionally, a special attribute, `compData` , may be defined on your `Options` subclass in order to provide more information to the shell-completion system. The attribute should be an instance of I DON'T KNOW WHAT TO DO WITH THIS LINK!

In addition, `compData` may be defined on parent classes in your inheritance hiearchy. The information from each I DON'T KNOW WHAT TO DO WITH THIS LINK!

DirDBM: Directory-based Storage

dirdbm.DirDBM

twisted.persisted.dirdbm.DirDBM is a DBM-like storage system. That is, it stores mappings between keys and values, like a Python dictionary, except that it stores the values in files in a directory - each entry is a different file. The keys must always be strings, as are the values. Other than that, DirDBM objects act just like Python dictionaries.

DirDBM is useful for cases when you want to store small amounts of data in an organized fashion, without having to deal with the complexity of a RDBMS or other sophisticated database. It is simple, easy to use, cross-platform, and doesn't require any external C libraries, unlike Python's built-in DBM modules.

```
>>> from twisted.persisted import dirdbm
>>> d = dirdbm.DirDBM("/tmp/dir")
>>> d["librarian"] = "ook"
>>> d["librarian"]
'ook'
>>> d.keys()
['librarian']
>>> del d["librarian"]
>>> d.items()
[]
```

dirdbm.Shelf

Sometimes it is necessary to persist more complicated objects than strings. With some care, dirdbm.Shelf can transparently persist them. `Shelf` works exactly like `DirDBM` , except that the values (but not the keys) can be arbitrary picklable objects. However, notice that mutating an object after it has been stored in the `Shelf` has no effect on the Shelf. When mutating objects, it is necessary to explicitly store them back in the `Shelf` afterwards:

```
>>> from twisted.persisted import dirdbm
>>> d = dirdbm.Shelf("/tmp/dir2")
>>> d["key"] = [1, 2]
>>> d["key"]
[1, 2]
>>> l = d["key"]
>>> l.append(3)
>>> d["key"]
[1, 2]
>>> d["key"] = l
>>> d["key"]
[1, 2, 3]
```

Writing tests for Twisted code using Trial

Trial basics

Trial is Twisted's testing framework. It provides a library for writing test cases and utility functions for working with the Twisted environment in your tests, and a command-line utility for running your tests. Trial is built on the Python standard library's `unittest` module. For more information on how Trial finds tests, see the loadModule documentation.

To run all the Twisted tests, do:

```
$ python -m twisted.trial twisted
```

Refer to the Trial man page for other command-line options.

Trial directories

You might notice a new `_trial_temp` folder in the current working directory after Trial completes the tests. This folder is the working directory for the Trial process. It can be used by unit tests and allows them to write whatever data they like to disk, and not worry about polluting the current working directory.

Folders named `_trial_temp-<counter>` are created if two instances of Trial are run in parallel from the same directory, so as to avoid giving two different test-runs the same temporary directory.

The twisted.python.lockfile utility is used to lock the `_trial_temp` directories. On Linux, this results in symlinks to pids. On Windows, directories are created with a single file with a pid as the contents. These lock files will be cleaned up if Trial exits normally and otherwise they will be left behind. They should be cleaned up the next time Trial tries to use the directory they lock, but it's also safe to delete them manually if desired.

Twisted-specific quirks: reactor, Deferreds, callLater

The standard Python `unittest` framework, from which Trial is derived, is ideal for testing code with a fairly linear flow of control. Twisted is an asynchronous networking framework which provides a clean, sensible way to establish functions that are run in response to events (like timers and incoming data), which creates a highly non-linear flow of control. Trial has a few extensions which help to test this kind of code. This section provides some hints on how to use these extensions and how to best structure your tests.

Leave the Reactor as you found it

Trial runs the entire test suite (over four thousand tests) in a single process, with a single reactor. Therefore it is important that your test leave the reactor in the same state as it found it. Leftover timers may expire during somebody else's unsuspecting test. Leftover connection attempts may complete (and fail) during a later test. These lead to intermittent failures that wander from test to test and are very time-consuming to track down.

If your test leaves event sources in the reactor, Trial will fail the test. The `tearDown` method is a good place to put cleanup code: it is always run regardless of whether your test passes or fails (like a `finally` clause in a try-except-finally construct). Exceptions in `tearDown` are flagged as errors and flunk the test. TestCase.addCleanup is another useful tool for cleaning up. With it, you can register callables to clean up resources as the test allocates them. Generally, code should be written so that only resources allocated in the tests need to be cleaned up in the tests. Resources which are allocated internally by the implementation should be cleaned up by the implementation.

If your code uses Deferreds or depends on the reactor running, you can return a Deferred from your test method, setUp, or tearDown and Trial will do the right thing. That is, it will run the reactor for you until the Deferred has triggered

and its callbacks have been run. Don't use `reactor.run()` , `reactor.stop()` , `reactor.crash()` or `reactor.iterate()` in your tests.

Calls to `reactor.callLater` create IDelayedCall s. These need to be run or cancelled during a test, otherwise they will outlive the test. This would be bad, because they could interfere with a later test, causing confusing failures in unrelated tests! For this reason, Trial checks the reactor to make sure there are no leftover IDelayedCall s in the reactor after a test, and will fail the test if there are. The cleanest and simplest way to make sure this all works is to return a Deferred from your test.

Similarly, sockets created during a test should be closed by the end of the test. This applies to both listening ports and client connections. So, calls to `reactor.listenTCP` (and `listenUNIX` , and so on) return IListeningPort s, and these should be cleaned up before a test ends by calling their stopListening method. Calls to `reactor.connectTCP` return IConnector s, which should be cleaned up by calling their disconnect method. Trial will warn about unclosed sockets.

The golden rule is: If your tests call a function which returns a Deferred, your test should return a Deferred.

Using Timers to Detect Failing Tests

It is common for tests to establish some kind of fail-safe timeout that will terminate the test in case something unexpected has happened and none of the normal test-failure paths are followed. This timeout puts an upper bound on the time that a test can consume, and prevents the entire test suite from stalling because of a single test. This is especially important for the Twisted test suite, because it is run automatically by the buildbot whenever changes are committed to the Git repository.

The way to do this in Trial is to set the `.timeout` attribute on your unit test method. Set the attribute to the number of seconds you wish to elapse before the test raises a timeout error. Trial has a default timeout which will be applied even if the `timeout` attribute is not set. The Trial default timeout is usually sufficient and should be overridden only in unusual cases.

Interacting with warnings in tests

Trial includes specific support for interacting with Python's `warnings` module. This support allows warning-emitting code to be written test-driven, just as any other code would be. It also improves the way in which warnings reporting when a test suite is running.

TestCase.flushWarnings allows tests to be written which make assertions about what warnings have been emitted during a particular test method. In order to test a warning with `flushWarnings` , write a test which first invokes the code which will emit a warning and then calls `flushWarnings` and makes assertions about the result. For example:

```
class SomeWarningsTests(TestCase):
    def test_warning(self):
        warnings.warn("foo is bad")
        self.assertEqual(len(self.flushWarnings()), 1)
```

Warnings emitted in tests which are not flushed will be included by the default reporter in its output after the result of the test. If Python's warnings filter system (see the-W command option to Python) is configured to treat a warning as an error, then unflushed warnings will causes tests to fail and will be included in the summary section of the default reporter. Note that unlike usual operation, when `warnings.warn` is called as part of a test method, it will not raise an exception when warnings have been configured as errors. However, if called outside of a test method (for example, at module scope in a test module or a module imported by a test module) then it *will* raise an exception.

Parallel test

In many situations, your unit tests may run faster if they are allowed to run in parallel, such that blocking I/O calls allow other tests to continue. Trial, like unittest, supports the -j parameter. Run `trial -j 3` to run 3 test runners at the same time.

This requires care in your test creation. Obviously, you need to ensure that your code is otherwise content to work in a parallel fashion while working within Twisted... and if you are using weird global variables in places, parallel tests might reveal this.

However, if you have a test that fires up a schema on an external database in the `setUp` function, does some operations on it in the test, and then deletes that schema in the tearDown function, your tests will behave in an unpredictable fashion as they tromp upon each other if they have their own schema. And this won't actually indicate a real error in your code, merely a testing-specific race-condition.

Extremely Low-Level Socket Operations

Introduction

Beyond supporting streams of data (SOCK_STREAM) or datagrams (SOCK_DGRAM), POSIX sockets have additional features not accessible via send(2) and recv(2). These features include things like scatter/gather I/O, duplicating file descriptors into other processes, and accessing out-of-band data.

Twisted includes a wrapper around the two C APIs which make these things possible, sendmsg and recvmsg . This document covers their usage. It is intended for Twisted maintainers. Application developers looking for this functionality should look for the high-level APIs Twisted provides on top of these wrappers.

sendmsg

`sendmsg(2)` exposes nearly all sender-side functionality of a socket. For a SOCK_STREAM socket, it can send bytes that become part of the stream of data being carried over the connection. For a SOCK_DGRAM socket, it can send bytes that become datagrams sent from the socket. It can send data from multiple memory locations (gather I/O). Over AF_UNIX sockets, it can copy file descriptors into whichever process is receiving on the other side. The wrapper included in Twisted, sendmsg, exposes many (but not all) of these features. This document covers the usage of the features it does expose. The primary limitation of this wrapper is that the interface supports sending only one *iovec* at a time.

recvmsg

Likewise, `recvmsg(2)` exposes nearly all the receiver-side functionality of a socket. It can receive stream data over from a SOCK_STREAM socket or datagrams from a SOCK_DGRAM socket. It can receive that data into multiple memory locations (scatter I/O), and it can receive those copied file descriptors. The wrapper included in Twisted, recvmsg, exposes many (but not all) of these features. This document covers the usage of the features it does expose. The primary limitation of this wrapper is that the interface supports receiving only one *iovec* at a time.

Sending And Receiving Regular Data

sendmsg can be used in a way which makes it equivalent to using the send call. The first argument to sendmsg is (in this case and all others) a socket over which to send the data. The second argument is a bytestring giving the data to send.

On the other end, recvmsg can be used to replace a recv call. The first argument to recvmsg is (again, in all cases) a socket over which to receive the data. The second argument is an integer giving the maximum number of bytes of data to receive.

send_replacement.py

```
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

"""
Demonstration of sending bytes over a TCP connection using sendmsg.
"""

from __future__ import print_function

from socket import socketpair

from twisted.python.sendmsg import sendmsg, recvmsg

def main():
    foo, bar = socketpair()
    sent = sendmsg(foo, b"Hello, world")
    print("Sent", sent, "bytes")
    (received, ancillary, flags) = recvmsg(bar, 1024)
    print("Received", repr(received))
    print("Extra stuff, boring in this case", flags, ancillary)

if __name__ == '__main__':
    main()
```

Copying File Descriptors

Used with an AF_UNIX socket, sendmsg send a copy of a file descriptor into whatever process is receiving on the other end of the socket. This is done using the ancillary data argument. Ancillary data consists of a list of three-tuples. A three-tuple constructed with SOL_SOCKET, SCM_RIGHTS, and a platform-endian packed file descriptor number will copy that file descriptor.

File descriptors copied this way must be received using a recvmsg call. No special arguments are required to receive these descriptors. They will appear, encoded as a native-order string, in the ancillary data list returned by recvmsg.

copy_descriptor.py

```
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

"""
Demonstration of copying a file descriptor over an AF_UNIX connection using
sendmsg.
"""

from __future__ import print_function

from os import pipe, read, write
from socket import SOL_SOCKET, socketpair
from struct import unpack, pack

from twisted.python.sendmsg import SCM_RIGHTS, sendmsg, recvmsg
```

```
def main():
    foo, bar = socketpair()
    reader, writer = pipe()

    # Send a copy of the descriptor.  Notice that there must be at least one
    # byte of normal data passed in.
    sent = sendmsg(
        foo, b"\x00", [(SOL_SOCKET, SCM_RIGHTS, pack("i", reader))])

    # Receive the copy, including that one byte of normal data.
    data, ancillary, flags = recvmsg(bar, 1024)
    duplicate = unpack("i", ancillary[0][2])[0]

    # Demonstrate that the copy works just like the original
    write(writer, b"Hello, world")
    print("Read from original (%d): %r" % (reader, read(reader, 6)))
    print("Read from duplicate (%d): %r" % (duplicate, read(duplicate, 6)))

if __name__ == '__main__':
    main()
```

A synchronous **M** essaging **P** rotocol Overview

A synchronous **M** essaging **P** rotocol Overview The purpose of this guide is to describe the uses for and usage of twisted.protocols.amp beyond what is explained in the API documentation. It will show you how to implement an AMP server which can respond to commands or interact directly with individual messages. It will also show you how to implement an AMP client which can issue commands to a server.

AMP is a bidirectional command/response-oriented protocol intended to be extended with application-specific request types and handlers. Various simple data types are supported and support for new data types can be added by applications.

Setting Up

AMP runs over a stream-oriented connection-based protocol, such as TCP or SSL. Before you can use any features of the AMP protocol, you need a connection. The protocol class to use to establish an AMP connection is AMP . Connection setup works as it does for almost all protocols in Twisted. For example, you can set up a listening AMP server using a server endpoint:

basic_server.tac

```
from twisted.protocols.amp import AMP
from twisted.internet import reactor
from twisted.internet.protocol import Factory
from twisted.internet.endpoints import TCP4ServerEndpoint
from twisted.application.service import Application
from twisted.application.internet import StreamServerEndpointService

application = Application("basic AMP server")

endpoint = TCP4ServerEndpoint(reactor, 8750)
factory = Factory()
factory.protocol = AMP
service = StreamServerEndpointService(endpoint, factory)
service.setServiceParent(application)
```

And you can connect to an AMP server using a client endpoint:

basic_client.py

```
if __name__ == '__main__':
    import basic_client
    raise SystemExit(basic_client.main())

from sys import stdout

from twisted.python.log import startLogging, err
from twisted.protocols.amp import AMP
from twisted.internet import reactor
from twisted.internet.protocol import Factory
from twisted.internet.endpoints import TCP4ClientEndpoint

def connect():
    endpoint = TCP4ClientEndpoint(reactor, "127.0.0.1", 8750)
    return endpoint.connect(Factory.forProtocol(AMP))


def main():
    startLogging(stdout)

    d = connect()
    d.addErrback(err, "Connection failed")
    def done(ignored):
        reactor.stop()
    d.addCallback(done)

    reactor.run()
```

Commands

Either side of an AMP connection can issue a command to the other side. Each kind of command is represented as a subclass of Command . A `Command` defines arguments, response values, and error conditions.

```
from twisted.protocols.amp import Integer, String, Unicode, Command

class UsernameUnavailable(Exception):
    pass

class RegisterUser(Command):
    arguments = [('username', Unicode()),
                 ('publickey', String())]

    response = [('uid', Integer())]

    errors = {UsernameUnavailable: 'username-unavailable'}
```

The definition of the command's signature - its arguments, response, and possible error conditions - is separate from the implementation of the behavior to execute when the command is received. The `Command` subclass only defines the former.

Commands are issued by calling `callRemote` on either side of the connection. This method returns a `Deferred` which eventually fires with the result of the command.

command_client.py

```
from __future__ import print_function

if __name__ == '__main__':
    import command_client
    raise SystemExit(command_client.main())

from sys import stdout

from twisted.python.log import startLogging, err
from twisted.protocols.amp import Integer, String, Unicode, Command
from twisted.internet import reactor

from basic_client import connect

class UsernameUnavailable(Exception):
    pass


class RegisterUser(Command):
    arguments = [('username', Unicode()),
                 ('publickey', String())]

    response = [('uid', Integer())]

    errors = {UsernameUnavailable: 'username-unavailable'}


def main():
    startLogging(stdout)

    d = connect()
    def connected(protocol):
        return protocol.callRemote(
            RegisterUser,
            username=u'alice',
            publickey='ssh-rsa AAAAB3NzaC1yc2 alice@actinium')
    d.addCallback(connected)

    def registered(result):
        print('Registration result:', result)
    d.addCallback(registered)

    d.addErrback(err, "Failed to register")

    def finished(ignored):
        reactor.stop()
    d.addCallback(finished)

    reactor.run()
```

Locators

The logic for handling a command can be specified as an object separate from the `AMP` instance which interprets and formats bytes over the network.

```
from twisted.protocols.amp import CommandLocator
from twisted.python.filepath import FilePath

class UsernameUnavailable(Exception):
    pass

class UserRegistration(CommandLocator):
    uidCounter = 0

    @RegisterUser.responder
    def register(self, username, publickey):
        path = FilePath(username)
        if path.exists():
            raise UsernameUnavailable()
        self.uidCounter += 1
        path.setContent('%d %s\n' % (self.uidCounter, publickey))
        return self.uidCounter
```

When you define a separate `CommandLocator` subclass, use it by passing an instance of it to the `AMP` initializer.

```
factory = Factory()
factory.protocol = lambda: AMP(locator=UserRegistration())
```

If no locator is passed in, `AMP` acts as its own locator. Command responders can be defined on an `AMP` subclass, just as the responder was defined on the `UserRegistration` example above.

Box Receivers

AMP conversations consist of an exchange of messages called *boxes* . A *box* consists of a sequence of pairs of key and value (for example, the pair `username` and `alice`). Boxes are generally represented as `dict` instances. Normally boxes are passed back and forth to implement the command request/response features described above. The logic for handling each box can be specified as an object separate from the `AMP` instance.

```
from zope.interface import implementer

from twisted.protocols.amp import IBoxReceiver

@implementer(IBoxReceiver)
class BoxReflector(object):
    def startReceivingBoxes(self, boxSender):
        self.boxSender = boxSender

    def ampBoxReceived(self, box):
        self.boxSender.sendBox(box)

    def stopReceivingBoxes(self, reason):
        self.boxSender = None
```

These methods parallel those of `IProtocol` . Startup notification is given by `startReceivingBoxes` . The argument passed to it is an `IBoxSender` provider, which can be used to send boxes back out over the network. `ampBoxReceived` delivers notification for a complete box having been received. And last, `stopReceivingBoxes` notifies the object that no more boxes will be received and no more can be sent. The argument passed to it is a `Failure` which may contain details about what caused the conversation to end.

To use a custom `IBoxReceiver` , pass it to the `AMP` initializer.

```
factory = Factory()
factory.protocol = lambda: AMP(boxReceiver=BoxReflector())
```

If no box receiver is passed in, `AMP` acts as its own box receiver. It handles boxes by treating them as command requests or responses and delivering them to the appropriate responder or as a result to a `callRemote` `Deferred`.

Overview of Twisted Spread

Perspective Broker (affectionately known as "PB") is an asynchronous, symmetric[1] network protocol for secure, remote method calls and transferring of objects. PB is "translucent, not transparent" , meaning that it is very visible and obvious to see the difference between local method calls and potentially remote method calls, but remote method calls are still extremely convenient to make, and it is easy to emulate them to have objects which work both locally and remotely.

PB supports user-defined serialized data in return values, which can be either copied each time the value is returned, or "cached" : only copied once and updated by notifications.

PB gets its name from the fact that access to objects is through a "perspective" . This means that when you are responding to a remote method call, you can establish who is making the call.

Rationale

No other currently existing protocols have all the properties of PB at the same time. The particularly interesting combination of attributes, though, is that PB is flexible and lightweight, allowing for rapid development, while still powerful enough to do two-way method calls and user-defined data types.

It is important to have these attributes in order to allow for a protocol which is extensible. One of the facets of this flexibility is that PB can integrate an arbitrary number of services could be aggregated over a single connection, as well as publish and call new methods on existing objects without restarting the server or client.

Introduction to Perspective Broker

Introduction

Suppose you find yourself in control of both ends of the wire: you have two programs that need to talk to each other, and you get to use any protocol you want. If you can think of your problem in terms of objects that need to make method calls on each other, then chances are good that you can use Twisted's Perspective Broker protocol rather than trying to shoehorn your needs into something like HTTP, or implementing yet another RPC mechanism[1] .

The Perspective Broker system (abbreviated "PB" , spawning numerous sandwich-related puns) is based upon a few central concepts:

- *serialization* : taking fairly arbitrary objects and types, turning them into a chunk of bytes, sending them over a wire, then reconstituting them on the other end. By keeping careful track of object ids, the serialized objects can contain references to other objects and the remote copy will still be useful.
- *remote method calls* : doing something to a local object and causing a method to get run on a distant one. The local object is called a RemoteReference , and you "do something" by running its `.callRemote` method.

[1] There is a negotiation phase for the `banana` serialization protocol with particular roles for listener and initiator, so it's not *completely* symmetric, but after the connection is fully established, the protocol is completely symmetrical.

[1] Most of Twisted is like this. Hell, most of Unix is like this: if *you* think it would be useful, someone else has probably thought that way in the past, and acted on it, and you can take advantage of the tool they created to solve the same problem you're facing now.

This document will contain several examples that will (hopefully) appear redundant and verbose once you've figured out what's going on. To begin with, much of the code will just be labelled "magic" : don't worry about how these parts work yet. It will be explained more fully later.

Object Roadmap

To start with, here are the major classes, interfaces, and functions involved in PB, with links to the file where they are defined (all of which are under twisted/, of course). Don't worry about understanding what they all do yet: it's easier to figure them out through their interaction than explaining them one at a time.

- Factory : `internet/protocol.py`
- PBServerFactory : `spread/pb.py`
- Broker : `spread/pb.py`

Other classes that are involved at some point:

- RemoteReference : `spread/pb.py`
- pb.Root : `spread/pb.py` , actually defined as `twisted.spread.flavors.Root` in `spread/flavors.py`
- pb.Referenceable : `spread/pb.py` , actually defined as `twisted.spread.flavors.Referenceable` in `spread/flavors.py`

Classes and interfaces that get involved when you start to care about authorization and security:

- Portal : `cred/portal.py`
- IRealm : `cred/portal.py`
- IPerspective : `spread/pb.py` , which you will usually be interacting with via pb.Avatar (a basic implementor of the interface).

Subclassing and Implementing

Technically you can subclass anything you want, but technically you could also write a whole new framework, which would just waste a lot of time. Knowing which classes are useful to subclass or which interfaces to implement is one of the bits of knowledge that's crucial to using PB (and all of Twisted) successfully. Here are some hints to get started:

- pb.Root , pb.Referenceable : you'll subclass these to make remotely-referenceable objects (i.e., objects which you can call methods on remotely) using PB. You don't need to change any of the existing behavior, just inherit all of it and add the remotely-accessible methods that you want to export.
- pb.Avatar : You'll be subclassing this when you get into PB programming with authorization. This is an implementor of IPerspective.
- ICredentialsChecker : Implement this if you want to authenticate your users against some sort of data store: i.e., an LDAP database, an RDBMS, etc. There are already a few implementations of this for various back-ends in twisted.cred.checkers.

Things you can Call Remotely

At this writing, there are three "flavors" of objects that can be accessed remotely through RemoteReference objects. Each of these flavors has a rule for how the `callRemote` message is transformed into a local method call on the server. In order to use one of these "flavors" , subclass them and name your published methods with the appropriate prefix.

- twisted.spread.pb.IPerspective implementors

  This is the first interface we deal with. It is a "perspective" onto your PB application. Perspectives are slightly special because they are usually the first object that a given user can access in your application (after they log on). A user should only receive a reference to their *own* perspective. PB works hard to verify, as best it can, that any method that can be called on a perspective directly is being called on behalf of the user who is represented by that perspective. (Services with unusual requirements for "on behalf of" , such as simulations with the ability to posses another player's avatar, are accomplished by providing indirected access to another user's perspective.)

  Perspectives are not usually serialized as remote references, so do not return an IPerspective-implementor directly.

  The way most people will want to implement IPerspective is by subclassing pb.Avatar. Remotely accessible methods on pb.Avatar instances are named with the `perspective_` prefix.

- twisted.spread.pb.Referenceable

  Referenceable objects are the simplest kind of PB object. You can call methods on them and return them from methods to provide access to other objects' methods.

  However, when a method is called on a Referenceable, it's not possible to tell who called it.

  Remotely accessible methods on Referenceables are named with the `remote_` prefix.

- twisted.spread.pb.Viewable

  Viewable objects are remotely referenceable objects which have the additional requirement that it must be possible to tell who is calling them. The argument list to a Viewable's remote methods is modified in order to include the Perspective representing the calling user.

  Remotely accessible methods on Viewables are named with the `view_` prefix.

Things you can Copy Remotely

In addition to returning objects that you can call remote methods on, you can return structured copies of local objects.

There are 2 basic flavors that allow for copying objects remotely. Again, you can use these by subclassing them. In order to specify what state you want to have copied when these are serialized, you can either use the Python default `__getstate__` or specialized method calls for that flavor.

- twisted.spread.pb.Copyable

  This is the simpler kind of object that can be copied. Every time this object is returned from a method or passed as an argument, it is serialized and unserialized.

  Copyable provides a method you can override, `getStateToCopyFor(perspective)` , which allows you to decide what an object will look like for the perspective who is requesting it. The `perspective` argument will be the perspective which is either passing an argument or returning a result an instance of your Copyable class.

  For security reasons, in order to allow a particular Copyable class to actually be copied, you must declare a `RemoteCopy` handler for that Copyable subclass. The easiest way to do this is to declare both in the same module, like so:

```
from twisted.spread import flavors
class Foo(flavors.Copyable):
    pass
class RemoteFoo(flavors.RemoteCopy):
    pass
flavors.setUnjellyableForClass(Foo, RemoteFoo)
```

In this case, each time a Foo is copied between peers, a RemoteFoo will be instantiated and populated with the Foo's state. If you do not do this, PB will complain that there have been security violations, and it may close the connection.

- twisted.spread.pb.Cacheable

  Let me preface this with a warning: Cacheable may be hard to understand. The motivation for it may be unclear if you don't have some experience with real-world applications that use remote method calling of some kind. Once you understand why you need it, what it does will likely seem simple and obvious, but if you get confused by this, forget about it and come back later. It's possible to use PB without understanding Cacheable at all.

  Cacheable is a flavor which is designed to be copied only when necessary, and updated on the fly as changes are made to it. When passed as an argument or a return value, if a Cacheable exists on the side of the connection it is being copied to, it will be referred to by ID and not copied.

  Cacheable is designed to minimize errors involved in replicating an object between multiple servers, especially those related to having stale information. In order to do this, Cacheable automatically registers observers and queries state atomically, together. You can override the method `getStateToCacheAndObserveFor(self, perspective, observer)` in order to specify how your observers will be stored and updated.

  Similar to `getStateToCopyFor` , `getStateToCacheAndObserveFor` gets passed a perspective. It also gets passed an `observer` , which is a remote reference to a "secret" fourth referenceable flavor: RemoteCache .

  A RemoteCache is simply the object that represents your Cacheable on the other side of the connection. It is registered using the same method as RemoteCopy , above. RemoteCache is different, however, in that it will be referenced by its peer. It acts as a Referenceable, where all methods prefixed with `observe_` will be callable remotely. It is recommended that your object maintain a list (note: library support for this is forthcoming!) of observers, and update them using `callRemote` when the Cacheable changes in a way that should be noticeable to its clients.

  Finally, when all references to a Cacheable from a given perspective are lost, `stoppedObserving(perspective, observer)` will be called on the Cacheable , with the same perspective/observer pair that `getStateToCacheAndObserveFor` was originally called with. Any cleanup remote calls can be made there, as well as removing the observer object from any lists which it was previously in. Any further calls to this observer object will be invalid.

Using Perspective Broker

Basic Example

The first example to look at is a complete (although somewhat trivial) application. It uses `PBServerFactory()` on the server side, and `PBClientFactory()` on the client side.

`pbsimple.py`

```
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from twisted.spread import pb
from twisted.internet import reactor

class Echoer(pb.Root):
    def remote_echo(self, st):
```

```
        print('echoing:', st)
        return st

if __name__ == '__main__':
    reactor.listenTCP(8789, pb.PBServerFactory(Echoer()))
    reactor.run()
```

pbsimpleclient.py

```
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.


from twisted.spread import pb
from twisted.internet import reactor
from twisted.python import util

factory = pb.PBClientFactory()
reactor.connectTCP("localhost", 8789, factory)
d = factory.getRootObject()
d.addCallback(lambda object: object.callRemote("echo", "hello network"))
d.addCallback(lambda echo: 'server echoed: '+echo)
d.addErrback(lambda reason: 'error: '+str(reason.value))
d.addCallback(util.println)
d.addCallback(lambda _: reactor.stop())
reactor.run()
```

First we look at the server. This defines an Echoer class (derived from pb.Root), with a method called `remote_echo()` . pb.Root objects (because of their inheritance of pb.Referenceable , described later) can define methods with names of the form `remote_*` ; a client which obtains a remote reference to that pb.Root object will be able to invoke those methods.

The pb.Root -ish object is given to a pb.PBServerFactory `()` . This is a Factory object like any other: the Protocol objects it creates for new connections know how to speak the PB protocol. The object you give to `pb.PBServerFactory()` becomes the "root object" , which simply makes it available for the client to retrieve. The client may only request references to the objects you want to provide it: this helps you implement your security model. Because it is so common to export just a single object (and because a `remote_*` method on that one can return a reference to any other object you might want to give out), the simplest example is one where the PBServerFactory is given the root object, and the client retrieves it.

The client side uses pb.PBClientFactory to make a connection to a given port. This is a two-step process involving opening a TCP connection to a given host and port and requesting the root object using `.getRootObject()` .

Because `.getRootObject()` has to wait until a network connection has been made and exchange some data, it may take a while, so it returns a Deferred, to which the gotObject() callback is attached. (See the documentation on *Deferring Execution* for a complete explanation of Deferred s). If and when the connection succeeds and a reference to the remote root object is obtained, this callback is run. The first argument passed to the callback is a remote reference to the distant root object. (you can give other arguments to the callback too, see the other parameters for `.addCallback()` and `.addCallbacks()`).

The callback does:

```
object.callRemote("echo", "hello network")
```

which causes the server's `.remote_echo()` method to be invoked. (running `.callRemote("boom")` would cause `.remote_boom()` to be run, etc). Again because of the delay involved, `callRemote()` returns a Deferred . Assuming the remote method was run without causing an exception (including an attempt to invoke an unknown

method), the callback attached to that Deferred will be invoked with any objects that were returned by the remote method call.

In this example, the server's `Echoer` object has a method invoked, *exactly* as if some code on the server side had done:

```
echoer_object.remote_echo("hello network")
```

and from the definition of `remote_echo()` we see that this just returns the same string it was given: "hello network" .

From the client's point of view, the remote call gets another Deferred object instead of that string. `callRemote()` *always* returns a Deferred . This is why PB is described as a system for "translucent" remote method calls instead of "transparent" ones: you cannot pretend that the remote object is really local. Trying to do so (as some other RPC mechanisms do, coughCORBAcough) breaks down when faced with the asynchronous nature of the network. Using Deferreds turns out to be a very clean way to deal with the whole thing.

The remote reference object (the one given to `getRootObject()` 's success callback) is an instance the RemoteReference class. This means you can use it to invoke methods on the remote object that it refers to. Only instances of RemoteReference are eligible for `.callRemote()` . The RemoteReference object is the one that lives on the remote side (the client, in this case), not the local side (where the actual object is defined).

In our example, the local object is that `Echoer()` instance, which inherits from pb.Root , which inherits from pb.Referenceable . It is that `Referenceable` class that makes the object eligible to be available for remote method calls[1] . If you have an object that is Referenceable, then any client that manages to get a reference to it can invoke any `remote_*` methods they please.

Note: The *only* thing they can do is invoke those methods. In particular, they cannot access attributes. From a security point of view, you control what they can do by limiting what the `remote_*` methods can do.

Also note: the other classes like Referenceable allow access to other methods, in particular `perspective_*` and `view_*` may be accessed. Don't write local-only methods with these names, because then remote callers will be able to do more than you intended.

Also also note: the other classes like pb.Copyable *do* allow access to attributes, but you control which ones they can see.

You don't have to be a pb.Root to be remotely callable, but you do have to be pb.Referenceable . (Objects that inherit from pb.Referenceable but not from pb.Root can be remotely called, but only pb.Root -ish objects can be given to the PBServerFactory .)

Complete Example

Here is an example client and server which uses pb.Referenceable as a root object and as the result of a remotely exposed method. In each context, methods can be invoked on the exposed Referenceable instance. In this example, the initial root object has a method that returns a reference to the second object.

`pb1server.py`

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.
```

[1] There are a few other classes that can bestow this ability, but pb.Referenceable is the easiest to understand; see 'flavors' below for details on the others.

```
from __future__ import print_function

from twisted.spread import pb

class Two(pb.Referenceable):
    def remote_three(self, arg):
        print("Two.three was given", arg)

class One(pb.Root):
    def remote_getTwo(self):
        two = Two()
        print("returning a Two called", two)
        return two

from twisted.internet import reactor
reactor.listenTCP(8800, pb.PBServerFactory(One()))
reactor.run()
```

pb1client.py

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from twisted.spread import pb
from twisted.internet import reactor

def main():
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 8800, factory)
    def1 = factory.getRootObject()
    def1.addCallbacks(got_obj1, err_obj1)
    reactor.run()

def err_obj1(reason):
    print("error getting first object", reason)
    reactor.stop()

def got_obj1(obj1):
    print("got first object:", obj1)
    print("asking it to getTwo")
    def2 = obj1.callRemote("getTwo")
    def2.addCallbacks(got_obj2)

def got_obj2(obj2):
    print("got second object:", obj2)
    print("telling it to do three(12)")
    obj2.callRemote("three", 12)

main()
```

pb.PBClientFactory.getRootObject will handle all the details of waiting for the creation of a connection. It returns a Deferred , which will have its callback called when the reactor connects to the remote server and pb.PBClientFactory gets the root, and have its `errback` called when the object-connection fails for any reason, whether it was host lookup failure, connection refusal, or some server-side error.

The root object has a method called `remote_getTwo` , which returns the `Two()` instance. On the client end, the callback gets a RemoteReference to that instance. The client can then invoke two's `.remote_three()` method.

RemoteReference objects have one method which is their purpose for being: `callRemote` . This method allows you to call a remote method on the object being referred to by the Reference. RemoteReference.callRemote , like pb.PBClientFactory.getRootObject , returns a Deferred . When a response to the method-call being sent arrives, the Deferred 's `callback` or `errback` will be made, depending on whether an error occurred in processing the method call.

You can use this technique to provide access to arbitrary sets of objects. Just remember that any object that might get passed "over the wire" must inherit from Referenceable (or one of the other flavors). If you try to pass a non-Referenceable object (say, by returning one from a `remote_*` method), you'll get an InsecureJelly exception[2] .

References can come back to you

If your server gives a reference to a client, and then that client gives the reference back to the server, the server will wind up with the same object it gave out originally. The serialization layer watches for returning reference identifiers and turns them into actual objects. You need to stay aware of where the object lives: if it is on your side, you do actual method calls. If it is on the other side, you do `.callRemote()`[3] .

`pb2server.py`

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from twisted.spread import pb
from twisted.internet import reactor

class Two(pb.Referenceable):
    def remote_print(self, arg):
        print("two.print was given", arg)

class One(pb.Root):
    def __init__(self, two):
        #pb.Root.__init__(self)    # pb.Root doesn't implement __init__
        self.two = two
    def remote_getTwo(self):
        print("One.getTwo(), returning my two called", self.two)
        return self.two
    def remote_checkTwo(self, newtwo):
        print("One.checkTwo(): comparing my two", self.two)
        print("One.checkTwo(): against your two", newtwo)
        if self.two == newtwo:
            print("One.checkTwo(): our twos are the same")


two = Two()
root_obj = One(two)
```

[2] This can be overridden, by subclassing one of the Serializable flavors and defining custom serialization code for your class. See *Passing Complex Types* for details.

[3] The binary nature of this local vs. remote scheme works because you cannot give RemoteReferences to a third party. If you could, then your object A could go to B, B could give it to C, C might give it back to you, and you would be hard pressed to tell if the object lived in C's memory space, in B's, or if it was really your own object, tarnished and sullied after being handed down like a really ugly picture that your great aunt owned and which nobody wants but which nobody can bear to throw out. Ok, not really like that, but you get the idea.

```
reactor.listenTCP(8800, pb.PBServerFactory(root_obj))
reactor.run()
```

pb2client.py

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from twisted.spread import pb
from twisted.internet import reactor

def main():
    foo = Foo()
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 8800, factory)
    factory.getRootObject().addCallback(foo.step1)
    reactor.run()

# keeping globals around is starting to get ugly, so we use a simple class
# instead. Instead of hooking one function to the next, we hook one method
# to the next.

class Foo:
    def __init__(self):
        self.oneRef = None

    def step1(self, obj):
        print("got one object:", obj)
        self.oneRef = obj
        print("asking it to getTwo")
        self.oneRef.callRemote("getTwo").addCallback(self.step2)

    def step2(self, two):
        print("got two object:", two)
        print("giving it back to one")
        print("one is", self.oneRef)
        self.oneRef.callRemote("checkTwo", two)

main()
```

The server gives a `Two()` instance to the client, who then returns the reference back to the server. The server compares the "two" given with the "two" received and shows that they are the same, and that both are real objects instead of remote references.

A few other techniques are demonstrated in `pb2client.py`. One is that the callbacks are added with `.addCallback` instead of `.addCallbacks`. As you can tell from the *Deferred* documentation, `.addCallback` is a simplified form which only adds a success callback. The other is that to keep track of state from one callback to the next (the remote reference to the main One() object), we create a simple class, store the reference in an instance thereof, and point the callbacks at a sequence of bound methods. This is a convenient way to encapsulate a state machine. Each response kicks off the next method, and any data that needs to be carried from one state to the next can simply be saved as an attribute of the object.

Remember that the client can give you back any remote reference you've given them. Don't base your zillion-dollar stock-trading clearinghouse server on the idea that you trust the client to give you back the right reference. The

security model inherent in PB means that they can *only* give you back a reference that you've given them for the current connection (not one you've given to someone else instead, nor one you gave them last time before the TCP session went down, nor one you haven't yet given to the client), but just like with URLs and HTTP cookies, the particular reference they give you is entirely under their control.

References to client-side objects

Anything that's Referenceable can get passed across the wire, *in either direction* . The "client" can give a reference to the "server" , and then the server can use .callRemote() to invoke methods on the client end. This fuzzes the distinction between "client" and "server" : the only real difference is who initiates the original TCP connection; after that it's all symmetric.

pb3server.py

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from twisted.spread import pb
from twisted.internet import reactor

class One(pb.Root):
    def remote_takeTwo(self, two):
        print("received a Two called", two)
        print("telling it to print(12)")
        two.callRemote("print", 12)

reactor.listenTCP(8800, pb.PBServerFactory(One()))
reactor.run()
```

pb3client.py

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from twisted.spread import pb
from twisted.internet import reactor

class Two(pb.Referenceable):
    def remote_print(self, arg):
        print("Two.print() called with", arg)

def main():
    two = Two()
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 8800, factory)
    def1 = factory.getRootObject()
    def1.addCallback(got_obj, two) # hands our 'two' to the callback
    reactor.run()
```

```
def got_obj(obj, two):
    print("got One:", obj)
    print("giving it our two")
    obj.callRemote("takeTwo", two)

main()
```

In this example, the client gives a reference to its own object to the server. The server then invokes a remote method on the client-side object.

Raising Remote Exceptions

Everything so far has covered what happens when things go right. What about when they go wrong? The Python Way is to raise an exception of some sort. The Twisted Way is the same.

The only special thing you do is to define your `Exception` subclass by deriving it from pb.Error . When any remotely-invokable method (like `remote_*` or `perspective_*`) raises a `pb.Error` -derived exception, a serialized form of that Exception object will be sent back over the wire[4] . The other side (which did `callRemote`) will have the "`errback`" callback run with a Failure object that contains a copy of the exception object. This `Failure` object can be queried to retrieve the error message and a stack traceback.

Failure is a special class, defined in `twisted/python/failure.py` , created to make it easier to handle asynchronous exceptions. Just as exception handlers can be nested, `errback` functions can be chained. If one errback can't handle the particular type of failure, it can be "passed along" to a errback handler further down the chain.

For simple purposes, think of the `Failure` as just a container for remotely-thrown `Exception` objects. To extract the string that was put into the exception, use its `.getErrorMessage()` method. To get the type of the exception (as a string), look at its `.type` attribute. The stack traceback is available too. The intent is to let the errback function get just as much information about the exception as Python's normal `try:` clauses do, even though the exception occurred in somebody else's memory space at some unknown time in the past.

`exc_server.py`

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from twisted.spread import pb
from twisted.internet import reactor

class MyError(pb.Error):
    """This is an Expected Exception. Something bad happened."""
    pass

class MyError2(Exception):
    """This is an Unexpected Exception. Something really bad happened."""
    pass

class One(pb.Root):
    def remote_broken(self):
        msg = "fall down go boom"
        print("raising a MyError exception with data '%s'" % msg)
```

[4] To be precise, the Failure will be sent if *any* exception is raised, not just pb.Error-derived ones. But the server will print ugly error messages if you raise ones that aren't derived from pb.Error.

```
        raise MyError(msg)
    def remote_broken2(self):
        msg = "hadda owie"
        print("raising a MyError2 exception with data '%s'" % msg)
        raise MyError2(msg)

def main():
    reactor.listenTCP(8800, pb.PBServerFactory(One()))
    reactor.run()

if __name__ == '__main__':
    main()
```

exc_client.py

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from twisted.spread import pb
from twisted.internet import reactor

def main():
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 8800, factory)
    d = factory.getRootObject()
    d.addCallbacks(got_obj)
    reactor.run()

def got_obj(obj):
    # change "broken" into "broken2" to demonstrate an unhandled exception
    d2 = obj.callRemote("broken")
    d2.addCallback(working)
    d2.addErrback(broken)

def working():
    print("erm, it wasn't *supposed* to work..")

def broken(reason):
    print("got remote Exception")
    # reason should be a Failure (or subclass) holding the MyError exception
    print(" .__class__ =", reason.__class__)
    print(" .getErrorMessage() =", reason.getErrorMessage())
    print(" .type =", reason.type)
    reactor.stop()

main()
```

```
$ ./exc_client.py
got remote Exception
 .__class__ = twisted.spread.pb.CopiedFailure
 .getErrorMessage() = fall down go boom
 .type = __main__.MyError
Main loop terminated.
```

Oh, and what happens if you raise some other kind of exception? Something that *isn't* subclassed from `pb.Error` ? Well, those are called "unexpected exceptions" , which make Twisted think that something has *really* gone wrong. These will raise an exception on the *server* side. This won't break the connection (the exception is trapped, just like most exceptions that occur in response to network traffic), but it will print out an unsightly stack trace on the server's stderr with a message that says "Peer Will Receive PB Traceback" , just as if the exception had happened outside a remotely-invokable method. (This message will go the current log target, if log.startLogging was used to redirect it). The client will get the same `Failure` object in either case, but subclassing your exception from `pb.Error` is the way to tell Twisted that you expect this sort of exception, and that it is ok to just let the client handle it instead of also asking the server to complain. Look at `exc_client.py` and change it to invoke `broken2()` instead of `broken()` to see the change in the server's behavior.

If you don't add an `errback` function to the Deferred , then a remote exception will still send a `Failure` object back over, but it will get lodged in the `Deferred` with nowhere to go. When that `Deferred` finally goes out of scope, the side that did `callRemote` will emit a message about an "Unhandled error in Deferred" , along with an ugly stack trace. It can't raise an exception at that point (after all, the `callRemote` that triggered the problem is long gone), but it will emit a traceback. So be a good programmer and *always* add `errback` handlers, even if they are just calls to log.err .

Try/Except blocks and Failure.trap

To implement the equivalent of the Python try/except blocks (which can trap particular kinds of exceptions and pass others "up" to higher-level `try/except` blocks), you can use the `.trap()` method in conjunction with multiple `errback` handlers on the `Deferred` . Re-raising an exception in an `errback` handler serves to pass that new exception to the next handler in the chain. The `trap` method is given a list of exceptions to look for, and will re-raise anything that isn't on the list. Instead of passing unhandled exceptions "up" to an enclosing `try` block, this has the effect of passing the exception "off" to later `errback` handlers on the same `Deferred` . The `trap` calls are used in chained errbacks to test for each kind of exception in sequence.

`trap_server.py`

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from twisted.internet import reactor
from twisted.spread import pb

class MyException(pb.Error):
    pass

class One(pb.Root):
    def remote_fooMethod(self, arg):
        if arg == "panic!":
            raise MyException
        return "response"
    def remote_shutdown(self):
        reactor.stop()

reactor.listenTCP(8800, pb.PBServerFactory(One()))
reactor.run()
```

`trap_client.py`

```
#!/usr/bin/env python
```

```
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from twisted.spread import pb, jelly
from twisted.python import log
from twisted.internet import reactor

class MyException(pb.Error): pass
class MyOtherException(pb.Error): pass

class ScaryObject:
    # not safe for serialization
    pass

def worksLike(obj):
    # the callback/errback sequence in class One works just like an
    # asynchronous version of the following:
    try:
        response = obj.callMethod(name, arg)
    except pb.DeadReferenceError:
        print(" stale reference: the client disconnected or crashed")
    except jelly.InsecureJelly:
        print(" InsecureJelly: you tried to send something unsafe to them")
    except (MyException, MyOtherException):
        print(" remote raised a MyException") # or MyOtherException
    except:
        print(" something else happened")
    else:
        print(" method successful, response:", response)

class One:
    def worked(self, response):
        print(" method successful, response:", response)
    def check_InsecureJelly(self, failure):
        failure.trap(jelly.InsecureJelly)
        print(" InsecureJelly: you tried to send something unsafe to them")
        return None
    def check_MyException(self, failure):
        which = failure.trap(MyException, MyOtherException)
        if which == MyException:
            print(" remote raised a MyException")
        else:
            print(" remote raised a MyOtherException")
        return None
    def catch_everythingElse(self, failure):
        print(" something else happened")
        log.err(failure)
        return None

    def doCall(self, explanation, arg):
        print(explanation)
        try:
            deferred = self.remote.callRemote("fooMethod", arg)
            deferred.addCallback(self.worked)
            deferred.addErrback(self.check_InsecureJelly)
            deferred.addErrback(self.check_MyException)
```

```
            deferred.addErrback(self.catch_everythingElse)
        except pb.DeadReferenceError:
            print(" stale reference: the client disconnected or crashed")

    def callOne(self):
        self.doCall("callOne: call with safe object", "safe string")
    def callTwo(self):
        self.doCall("callTwo: call with dangerous object", ScaryObject())
    def callThree(self):
        self.doCall("callThree: call that raises remote exception", "panic!")
    def callShutdown(self):
        print("telling them to shut down")
        self.remote.callRemote("shutdown")
    def callFour(self):
        self.doCall("callFour: call on stale reference", "dummy")

    def got_obj(self, obj):
        self.remote = obj
        reactor.callLater(1, self.callOne)
        reactor.callLater(2, self.callTwo)
        reactor.callLater(3, self.callThree)
        reactor.callLater(4, self.callShutdown)
        reactor.callLater(5, self.callFour)
        reactor.callLater(6, reactor.stop)

factory = pb.PBClientFactory()
reactor.connectTCP("localhost", 8800, factory)
deferred = factory.getRootObject()
deferred.addCallback(One().got_obj)
reactor.run()
```

```
$ ./trap_client.py
callOne: call with safe object
 method successful, response: response
callTwo: call with dangerous object
 InsecureJelly: you tried to send something unsafe to them
callThree: call that raises remote exception
 remote raised a MyException
telling them to shut down
callFour: call on stale reference
 stale reference: the client disconnected or crashed
```

In this example, `callTwo` tries to send an instance of a locally-defined class through `callRemote` . The default security model implemented by jelly on the remote end will not allow unknown classes to be unserialized (i.e. taken off the wire as a stream of bytes and turned back into an object: a living, breathing instance of some class): one reason is that it does not know which local class ought to be used to create an instance that corresponds to the remote object[5] .

The receiving end of the connection gets to decide what to accept and what to reject. It indicates its disapproval by raising a jelly.InsecureJelly exception. Because it occurs at the remote end, the exception is returned to the caller asynchronously, so an `errback` handler for the associated `Deferred` is run. That errback receives a `Failure`

[5] The naive approach of simply doing `import SomeClass` to match a remote caller who claims to have an object of type "SomeClass" could have nasty consequences for some modules that do significant operations in their `__init__` methods (think `telnetlib.Telnet(host='localhost', port='chargen')` , or even more powerful classes that you have available in your server program). Allowing a remote entity to create arbitrary classes in your namespace is nearly equivalent to allowing them to run arbitrary code.

The InsecureJelly exception arises because the class being sent over the wire has not been registered with the serialization layer (known as jelly). The easiest way to make it possible to copy entire class instances over the wire is to have them inherit from pb.Copyable , and then to use `setUnjellyableForClass(remoteClass, localClass)` on the receiving side. See *Passing Complex Types* for an example.

which wraps the `InsecureJelly` .

Remember that `trap` re-raises exceptions that it wasn't asked to look for. You can only check for one set of exceptions per errback handler: all others must be checked in a subsequent handler. `check_MyException` shows how multiple kinds of exceptions can be checked in a single errback: give a list of exception types to `trap` , and it will return the matching member. In this case, the kinds of exceptions we are checking for (`MyException` and `MyOtherException`) may be raised by the remote end: they inherit from pb.Error .

The handler can return `None` to terminate processing of the errback chain (to be precise, it switches to the callback that follows the errback; if there is no callback then processing terminates). It is a good idea to put an errback that will catch everything (no `trap` tests, no possible chance of raising more exceptions, always returns `None`) at the end of the chain. Just as with regular `try: except:` handlers, you need to think carefully about ways in which your errback handlers could themselves raise exceptions. The extra importance in an asynchronous environment is that an exception that falls off the end of the `Deferred` will not be signalled until that `Deferred` goes out of scope, and at that point may only cause a log message (which could even be thrown away if log.startLogging is not used to point it at stdout or a log file). In contrast, a synchronous exception that is not handled by any other `except:` block will very visibly terminate the program immediately with a noisy stack trace.

`callFour` shows another kind of exception that can occur while using `callRemote` : pb.DeadReferenceError . This one occurs when the remote end has disconnected or crashed, leaving the local side with a stale reference. This kind of exception happens to be reported right away (XXX: is this guaranteed? probably not), so must be caught in a traditional synchronous `try: except pb.DeadReferenceError` block.

Yet another kind that can occur is a pb.PBConnectionLost exception. This occurs (asynchronously) if the connection was lost while you were waiting for a `callRemote` call to complete. When the line goes dead, all pending requests are terminated with this exception. Note that you have no way of knowing whether the request made it to the other end or not, nor how far along in processing it they had managed before the connection was lost. XXX: explain transaction semantics, find a decent reference.

Managing Clients of Perspectives

Author Kevin Turner

Overview

In all the IPerspective uses we have shown so far, we ignored the `mind` argument and created a new `Avatar` for every connection. This is usually an easy design choice, and it works well for simple cases.

In more complicated cases, for example an `Avatar` that represents a player object which is persistent in the game universe, we will want connections from the same player to use the same `Avatar` .

Another thing which is necessary in more complicated scenarios is notifying a player asynchronously. While it is possible, of course, to allow a player to call `perspective_remoteListener(referencable)` that would mean both duplication of code and a higher latency in logging in, both bad.

In previous sections all realms looked to be identical. In this one we will show the usefulness of realms in accomplishing those two objectives.

Managing Avatars

The simplest way to manage persistent avatars is to use a straight-forward caching mechanism:

```
from zope.interface import implementer

class SimpleAvatar(pb.Avatar):
```

```
    greetings = 0
    def __init__(self, name):
        self.name = name
    def perspective_greet(self):
        self.greetings += 1
        return "<%d>hello %s" % (self.greetings, self.name)

@implementer(portal.IRealm)
class CachingRealm:

    def __init__(self):
        self.avatars = {}

    def requestAvatar(self, avatarId, mind, *interfaces):
        if pb.IPerspective not in interfaces: raise NotImplementedError
        if avatarId in self.avatars:
            p = self.avatars[avatarId]
        else:
            p = self.avatars[avatarId] = SimpleAvatar(avatarId)
        return pb.IPerspective, p, lambda:None
```

This gives us a perspective which counts the number of greetings it sent its client. Implementing a caching strategy, as opposed to generating a realm with the correct avatars already in it, is usually easier. This makes adding new checkers to the portal, or adding new users to a checker database, transparent. Otherwise, careful synchronization is needed between the checker and avatar is needed (much like the synchronization between UNIX's `/etc/shadow` and `/etc/passwd`).

Sometimes, however, an avatar will need enough per-connection state that it would be easier to generate a new avatar and cache something else. Here is an example of that:

```
from zope.interface import implementer

class Greeter:
    greetings = 0
    def hello(self):
        self.greetings += 1
        return "<%d>hello" % (self.greetings, self.name)

class SimpleAvatar(pb.Avatar):
    def __init__(self, name, greeter):
        self.name = name
        self.greeter = greeter
    def perspective_greet(self):
        return self.greeter.hello()+' '+self.name

@implementer(portal.IRealm)
class CachingRealm:
    def __init__(self):
        self.greeters = {}

    def requestAvatar(self, avatarId, mind, *interfaces):
        if pb.IPerspective not in interfaces: raise NotImplementedError
        if avatarId in self.greeters:
            p = self.greeters[avatarId]
        else:
            p = self.greeters[avatarId] = Greeter()
        return pb.IPerspective, SimpleAvatar(avatarId, p), lambda:None
```

It might seem tempting to use this pattern to have an avatar which is notified of new connections. However, the problems here are twofold: it would lead to a thin class which needs to forward all of its methods, and it would be impossible to know when disconnections occur. Luckily, there is a better pattern:

```
from zope.interface import implementer

class SimpleAvatar(pb.Avatar):
    greetings = 0
    connections = 0
    def __init__(self, name):
        self.name = name
    def connect(self):
        self.connections += 1
    def disconnect(self):
        self.connections -= 1
    def perspective_greet(self):
        self.greetings += 1
        return "<%d>hello %s" % (self.greetings, self.name)

@implementer(portal.IRealm)
class CachingRealm:
    def __init__(self):
        self.avatars = {}

    def requestAvatar(self, avatarId, mind, *interfaces):
        if pb.IPerspective not in interfaces: raise NotImplementedError
        if avatarId in self.avatars:
            p = self.avatars[avatarId]
        else:
            p = self.avatars[avatarId] = SimpleAvatar(avatarId)
        p.connect()
        return pb.IPerspective, p, p.disconnect
```

It is possible to use such a pattern to define an arbitrary limit for the number of concurrent connections:

```
from zope.interface import implementer

class SimpleAvatar(pb.Avatar):
    greetings = 0
    connections = 0
    def __init__(self, name):
        self.name = name
    def connect(self):
        self.connections += 1
    def disconnect(self):
        self.connections -= 1
    def perspective_greet(self):
        self.greetings += 1
        return "<%d>hello %s" % (self.greetings, self.name)

@implementer(portal.IRealm)
class CachingRealm:
    def __init__(self, max=1):
        self.avatars = {}
        self.max = max

    def requestAvatar(self, avatarId, mind, *interfaces):
        if pb.IPerspective not in interfaces: raise NotImplementedError
```

```
        if avatarId in self.avatars:
            p = self.avatars[avatarId]
        else:
            p = self.avatars[avatarId] = SimpleAvatar(avatarId)
        if p.connections >= self.max:
            raise ValueError("too many connections")
        p.connect()
        return pb.IPerspective, p, p.disconnect
```

Managing Clients

So far, all our realms have ignored the `mind` argument. In the case of PB, the `mind` is an object supplied by the remote login method – usually, when it passes over the wire, it becomes a `pb.RemoteReference` . This object allows sending messages to the client as soon as the connection is established and authenticated.

Here is a simple remote-clock application which shows the usefulness of the `mind` argument:

```
from zope.interface import implementer

class SimpleAvatar(pb.Avatar):
    def __init__(self, client):
        self.s = internet.TimerService(1, self.telltime)
        self.s.startService()
        self.client = client
    def telltime(self):
        self.client.callRemote("notifyTime", time.time())
    def perspective_setperiod(self, period):
        self.s.stopService()
        self.s = internet.TimerService(period, self.telltime)
        self.s.startService()
    def logout(self):
        self.s.stopService()

@implementer(portal.IRealm)
class Realm:
    def requestAvatar(self, avatarId, mind, *interfaces):
        if pb.IPerspective not in interfaces: raise NotImplementedError
        p = SimpleAvatar(mind)
        return pb.IPerspective, p, p.logout
```

In more complicated situations, you might want to cache the avatars and give each one a set of "current clients" or something similar.

PB Copyable: Passing Complex Types

Overview

This chapter focuses on how to use PB to pass complex types (specifically class instances) to and from a remote process. The first section is on simply copying the contents of an object to a remote process (pb.Copyable). The second covers how to copy those contents once, then update them later when they change (Cacheable).

Motivation

From the *previous chapter* , you've seen how to pass basic types to a remote process, by using them in the arguments or return values of a callRemote function. However, if you've experimented with it, you may have discovered problems when trying to pass anything more complicated than a primitive int/list/dict/string type, or another pb.Referenceable object. At some point you want to pass entire objects between processes, instead of having to reduce them down to dictionaries on one end and then re-instantiating them on the other.

Passing Objects

The most obvious and straightforward way to send an object to a remote process is with something like the following code. It also happens that this code doesn't work, as will be explained below.

```
class LilyPond:
  def __init__(self, frogs):
    self.frogs = frogs

pond = LilyPond(12)
ref.callRemote("sendPond", pond)
```

If you try to run this, you might hope that a suitable remote end which implements the `remote_sendPond` method would see that method get invoked with an instance from the `LilyPond` class. But instead, you'll encounter the dreaded InsecureJelly exception. This is Twisted's way of telling you that you've violated a security restriction, and that the receiving end refuses to accept your object.

Security Options

What's the big deal? What's wrong with just copying a class into another process' namespace?

Reversing the question might make it easier to see the issue: what is the problem with accepting a stranger's request to create an arbitrary object in your local namespace? The real question is how much power you are granting them: what actions can they convince you to take on the basis of the bytes they are sending you over that remote connection.

Objects generally represent more power than basic types like strings and dictionaries because they also contain (or reference) code, which can modify other data structures when executed. Once previously-trusted data is subverted, the rest of the program is compromised.

The built-in Python "batteries included" classes are relatively tame, but you still wouldn't want to let a foreign program use them to create arbitrary objects in your namespace or on your computer. Imagine a protocol that involved sending a file-like object with a `read()` method that was supposed to used later to retrieve a document. Then imagine what if that object were created with `os.fdopen("~/.gnupg/secring.gpg")` . Or an instance of `telnetlib.Telnet("localhost", "chargen")` .

Classes you've written for your own program are likely to have far more power. They may run code during `__init__` , or even have special meaning simply because of their existence. A program might have `User` objects to represent user accounts, and have a rule that says all `User` objects in the system are referenced when authorizing a login session. (In this system, `User.__init__` would probably add the object to a global list of known users). The simple act of creating an object would give access to somebody. If you could be tricked into creating a bad object, an unauthorized user would get access.

So object creation needs to be part of a system's security design. The dotted line between "trusted inside" and "untrusted outside" needs to describe what may be done in response to outside events. One of those events is the receipt of an object through a PB remote procedure call, which is a request to create an object in your "inside" namespace. The question is what to do in response to it. For this reason, you must explicitly specify what remote classes will be accepted, and how their local representatives are to be created.

What class to use?

Another basic question to answer before we can do anything useful with an incoming serialized object is: what class should we create? The simplistic answer is to create the "same kind" that was serialized on the sender's end of the wire, but this is not as easy or as straightforward as you might think. Remember that the request is coming from a different program, using a potentially different set of class libraries. In fact, since PB has also been implemented in Java, Emacs-Lisp, and other languages, there's no guarantee that the sender is even running Python! All we know on the receiving end is a list of two things which describe the instance they are trying to send us: the name of the class, and a representation of the contents of the object.

PB lets you specify the mapping from remote class names to local classes with the setUnjellyableForClass function[1] .

This function takes a remote/sender class reference (either the fully-qualified name as used by the sending end, or a class object from which the name can be extracted), and a local/recipient class (used to create the local representation for incoming serialized objects). Whenever the remote end sends an object, the class name that they transmit is looked up in the table controlled by this function. If a matching class is found, it is used to create the local object. If not, you get the `InsecureJelly` exception.

In general you expect both ends to share the same codebase: either you control the program that is running on both ends of the wire, or both programs share some kind of common language that is implemented in code which exists on both ends. You wouldn't expect them to send you an object of the MyFooziWhatZit class unless you also had a definition for that class. So it is reasonable for the Jelly layer to reject all incoming classes except the ones that you have explicitly marked with `setUnjellyableForClass` . But keep in mind that the sender's idea of a `User` object might differ from the recipient's, either through namespace collisions between unrelated packages, version skew between nodes that haven't been updated at the same rate, or a malicious intruder trying to cause your code to fail in some interesting or potentially vulnerable way.

pb.Copyable

Ok, enough of this theory. How do you send a fully-fledged object from one side to the other?

`copy_sender.py`

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from twisted.spread import pb, jelly
from twisted.python import log
from twisted.internet import reactor

class LilyPond:
    def setStuff(self, color, numFrogs):
        self.color = color
        self.numFrogs = numFrogs
    def countFrogs(self):
        print("%d frogs" % self.numFrogs)
```

[1] Note that, in this context, "unjelly" is a verb with the opposite meaning of "jelly" . The verb "to jelly" means to serialize an object or data structure into a sequence of bytes (or other primitive transmittable/storable representation), while "to unjelly" means to unserialize the bytestream into a live object in the receiver's memory space. "Unjellyable" is a noun, (*not* an adjective), referring to the class that serves as a destination or recipient of the unjellying process. "A is unjellyable into B" means that a serialized representation A (of some remote object) can be unserialized into a local object of type B. It is these objects "B" that are the "Unjellyable" second argument of the `setUnjellyableForClass` function. In particular, "unjellyable" does *not* mean "cannot be jellied" . Unpersistable means "not persistable" , but "unjelly" , "unserialize" , and "unpickle" mean to reverse the operations of "jellying" , "serializing" , and "pickling" .

```
class CopyPond(LilyPond, pb.Copyable):
    pass

class Sender:
    def __init__(self, pond):
        self.pond = pond

    def got_obj(self, remote):
        self.remote = remote
        d = remote.callRemote("takePond", self.pond)
        d.addCallback(self.ok).addErrback(self.notOk)

    def ok(self, response):
        print("pond arrived", response)
        reactor.stop()
    def notOk(self, failure):
        print("error during takePond:")
        if failure.type == jelly.InsecureJelly:
            print(" InsecureJelly")
        else:
            print(failure)
        reactor.stop()
        return None

def main():
    from copy_sender import CopyPond  # so it's not __main__.CopyPond
    pond = CopyPond()
    pond.setStuff("green", 7)
    pond.countFrogs()
    # class name:
    print(".".join([pond.__class__.__module__, pond.__class__.__name__]))

    sender = Sender(pond)
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 8800, factory)
    deferred = factory.getRootObject()
    deferred.addCallback(sender.got_obj)
    reactor.run()

if __name__ == '__main__':
    main()
```

copy_receiver.tac

```
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

"""
PB copy receiver example.

This is a Twisted Application Configuration (tac) file.  Run with e.g.
   twistd -ny copy_receiver.tac

See the twistd(1) man page or
http://twistedmatrix.com/documents/current/howto/application for details.
"""
```

```
from __future__ import print_function

import sys
if __name__ == '__main__':
    print(__doc__)
    sys.exit(1)

from twisted.application import service, internet
from twisted.internet import reactor
from twisted.spread import pb
from copy_sender import LilyPond, CopyPond

from twisted.python import log
#log.startLogging(sys.stdout)

class ReceiverPond(pb.RemoteCopy, LilyPond):
    pass
pb.setUnjellyableForClass(CopyPond, ReceiverPond)

class Receiver(pb.Root):
    def remote_takePond(self, pond):
        print(" got pond:", pond)
        pond.countFrogs()
        return "safe and sound" # positive acknowledgement
    def remote_shutdown(self):
        reactor.stop()

application = service.Application("copy_receiver")
internet.TCPServer(8800, pb.PBServerFactory(Receiver())).setServiceParent(
    service.IServiceCollection(application))
```

The sending side has a class called `LilyPond` . To make this eligible for transport through `callRemote` (either as an argument, a return value, or something referenced by either of those [like a dictionary value]), it must inherit from one of the four Serializable classes. In this section, we focus on Copyable . The copyable subclass of `LilyPond` is called `CopyPond` . We create an instance of it and send it through `callRemote` as an argument to the receiver's `remote_takePond` method. The Jelly layer will serialize ("jelly") that object as an instance with a class name of"copy_sender.CopyPond" and some chunk of data that represents the object's state. `pond.__class__.__module__` and `pond.__class__.__name__` are used to derive the class name string. The object's getStateToCopy method is used to get the state: this is provided by pb.Copyable , and the default just retrieves `self.__dict__` . This works just like the optional `__getstate__` method used by `pickle` . The pair of name and state are sent over the wire to the receiver.

The receiving end defines a local class named `ReceiverPond` to represent incoming `LilyPond` instances. This class derives from the sender's `LilyPond` class (with a fully-qualified name of `copy_sender.LilyPond`), which specifies how we expect it to behave. We trust that this is the same `LilyPond` class as the sender used. (At the very least, we hope ours will be able to accept a state created by theirs). It also inherits from pb.RemoteCopy , which is a requirement for all classes that act in this local-representative role (those which are given to the second argument of `setUnjellyableForClass`). `RemoteCopy` provides the methods that tell the Jelly layer how to create the local object from the incoming serialized state.

Then `setUnjellyableForClass` is used to register the two classes. This has two effects: instances of the remote class (the first argument) will be allowed in through the security layer, and instances of the local class (the second argument) will be used to contain the state that is transmitted when the sender serializes the remote object.

When the receiver unserializes ("unjellies") the object, it will create an instance of the local `ReceiverPond` class, and hand the transmitted state (usually in the form of a dictionary) to that object's setCopyableState method. This acts just like the `__setstate__` method that `pickle` uses when unserializing an object. `getStateToCopy`

/setCopyableState are distinct from __getstate__ /__setstate__ to allow objects to be persisted (across time) differently than they are transmitted (across [memory]space).

When this is run, it produces the following output:

```
[-] twisted.spread.pb.PBServerFactory starting on 8800
[-] Starting factory <twisted.spread.pb.PBServerFactory instance at
0x406159cc>
[Broker,0,127.0.0.1]  got pond: <__builtin__.ReceiverPond instance at
0x406ec5ec>
[Broker,0,127.0.0.1] 7 frogs
```

```
$ ./copy_sender.py
7 frogs
copy_sender.CopyPond
pond arrived safe and sound
Main loop terminated.
$
```

Controlling the Copied State

By overriding getStateToCopy and setCopyableState , you can control how the object is transmitted over the wire. For example, you might want perform some data-reduction: pre-compute some results instead of sending all the raw data over the wire. Or you could replace references to a local object on the sender's side with markers before sending, then upon receipt replace those markers with references to a receiver-side proxy that could perform the same operations against a local cache of data.

Another good use for getStateToCopy is to implement "local-only" attributes: data that is only accessible by the local process, not to any remote users. For example, a .password attribute could be removed from the object state before sending to a remote system. Combined with the fact that Copyable objects return unchanged from a round trip, this could be used to build a challenge-response system (in fact PB does this with pb.Referenceable objects to implement authorization as described *here*).

Whatever getStateToCopy returns from the sending object will be serialized and sent over the wire; setCopyableState gets whatever comes over the wire and is responsible for setting up the state of the object it lives in.

copy2_classes.py

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from twisted.spread import pb

class FrogPond:
    def __init__(self, numFrogs, numToads):
        self.numFrogs = numFrogs
        self.numToads = numToads
    def count(self):
        return self.numFrogs + self.numToads

class SenderPond(FrogPond, pb.Copyable):
    def getStateToCopy(self):
        d = self.__dict__.copy()
        d['frogsAndToads'] = d['numFrogs'] + d['numToads']
```

```
        del d['numFrogs']
        del d['numToads']
        return d

class ReceiverPond(pb.RemoteCopy):
    def setCopyableState(self, state):
        self.__dict__ = state
    def count(self):
        return self.frogsAndToads

pb.setUnjellyableForClass(SenderPond, ReceiverPond)
```

copy2_sender.py

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from twisted.spread import pb, jelly
from twisted.python import log
from twisted.internet import reactor
from copy2_classes import SenderPond

class Sender:
    def __init__(self, pond):
        self.pond = pond

    def got_obj(self, obj):
        d = obj.callRemote("takePond", self.pond)
        d.addCallback(self.ok).addErrback(self.notOk)

    def ok(self, response):
        print("pond arrived", response)
        reactor.stop()
    def notOk(self, failure):
        print("error during takePond:")
        if failure.type == jelly.InsecureJelly:
            print(" InsecureJelly")
        else:
            print(failure)
        reactor.stop()
        return None

def main():
    pond = SenderPond(3, 4)
    print("count %d" % pond.count())

    sender = Sender(pond)
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 8800, factory)
    deferred = factory.getRootObject()
    deferred.addCallback(sender.got_obj)
    reactor.run()

if __name__ == '__main__':
```

```
    main()
```

copy2_receiver.py

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from twisted.application import service, internet
from twisted.internet import reactor
from twisted.spread import pb
import copy2_classes # needed to get ReceiverPond registered with Jelly

class Receiver(pb.Root):
    def remote_takePond(self, pond):
        print(" got pond:", pond)
        print(" count %d" % pond.count())
        return "safe and sound" # positive acknowledgement
    def remote_shutdown(self):
        reactor.stop()

application = service.Application("copy_receiver")
internet.TCPServer(8800, pb.PBServerFactory(Receiver())).setServiceParent(
    service.IServiceCollection(application))
```

In this example, the classes are defined in a separate source file, which also sets up the binding between them. The `SenderPond` and `ReceiverPond` are unrelated save for this binding: they happen to implement the same methods, but use different internal instance variables to accomplish them.

The recipient of the object doesn't even have to import the class definition into their namespace. It is sufficient that they import the class definition (and thus execute the `setUnjellyableForClass` statement). The Jelly layer remembers the class definition until a matching object is received. The sender of the object needs the definition, of course, to create the object in the first place.

When run, the `copy2` example emits the following:

```
$ twistd -n -y copy2_receiver.py
[-] twisted.spread.pb.PBServerFactory starting on 8800
[-] Starting factory <twisted.spread.pb.PBServerFactory instance at
0x40604b4c>
[Broker,0,127.0.0.1]  got pond: <copy2_classes.ReceiverPond instance at
0x406eb2ac>
[Broker,0,127.0.0.1]  count 7
```

```
$ ./copy2_sender.py
count 7
pond arrived safe and sound
Main loop terminated.
```

Things To Watch Out For

- The first argument to `setUnjellyableForClass` must refer to the class *as known by the sender* . The sender has no way of knowing about how your local `import` statements are set up, and Python's flexible namespace semantics allow you to access the same class through a variety of different names. You must match whatever the sender does. Having both ends import the class from a separate file, using a canonical module name (no "sibling imports"), is a good way to get this right, especially when both the sending and the receiving classes are defined together, with the `setUnjellyableForClass` immediately following them.
- The class that is sent must inherit from pb.Copyable . The class that is registered to receive it must inherit from pb.RemoteCopy[2] .
- The same class can be used to send and receive. Just have it inherit from both `pb.Copyable` and `pb.RemoteCopy` . This will also make it possible to send the same class symmetrically back and forth over the wire. But don't get confused about when it is coming (and using `setCopyableState`) versus when it is going (using `getStateToCopy`).
- InsecureJelly exceptions are raised by the receiving end. They will be delivered asynchronously to an `errback` handler. If you do not add one to the `Deferred` returned by `callRemote` , then you will never receive notification of the problem.
- The class that is derived from pb.RemoteCopy will be created using a constructor `__init__` method that takes no arguments. All setup must be performed in the `setCopyableState` method. As the docstring on RemoteCopy says, don't implement a constructor that requires arguments in a subclass of `RemoteCopy` .

More Information

- `pb.Copyable` is mostly implemented in `twisted.spread.flavors` , and the docstrings there are the best source of additional information.
- `Copyable` is also used in twisted.web.distrib to deliver HTTP requests to other programs for rendering, allowing subtrees of URL space to be delegated to multiple programs (on multiple machines).

pb.Cacheable

Sometimes the object you want to send to the remote process is big and slow. "big" means it takes a lot of data (storage, network bandwidth, processing) to represent its state. "slow" means that state doesn't change very frequently. It may be more efficient to send the full state only once, the first time it is needed, then afterwards only send the differences or changes in state whenever it is modified. The pb.Cacheable class provides a framework to implement this.

pb.Cacheable is derived from pb.Copyable , so it is based upon the idea of an object's state being captured on the sending side, and then turned into a new object on the receiving side. This is extended to have an object "publishing" on the sending side (derived from pb.Cacheable), matched with one"observing" on the receiving side (derived from pb.RemoteCache).

To effectively use `pb.Cacheable` , you need to isolate changes to your object into accessor functions (specifically "setter" functions). Your object needs to get control *every* single time some attribute is changed[3] .

You derive your sender-side class from `pb.Cacheable` , and you add two methods: getStateToCacheAndObserveFor and stoppedObserving . The first is called when a remote caching reference is first created, and retrieves the data

[2] pb.RemoteCopy is actually defined in twisted.spread.flavors , but `pb.RemoteCopy` is the preferred way to access it

[3] Of course you could be clever and add a hook to `__setattr__` , along with magical change-announcing subclasses of the usual builtin types, to detect changes that result from normal "=" set operations. The semi-magical "property attributes" that were introduced in Python 2.2 could be useful too. The result might be hard to maintain or extend, though.

with which the cache is first filled. It also provides an object called the "observer"[4] that points at that receiver-side cache. Every time the state of the object is changed, you give a message to the observer, informing them of the change. The other method, `stoppedObserving` , is called when the remote cache goes away, so that you can stop sending updates.

On the receiver end, you make your cache class inherit from pb.RemoteCache , and implement the `setCopyableState` as you would for a `pb.RemoteCopy` object. In addition, you must implement methods to receive the updates sent to the observer by the `pb.Cacheable` : these methods should have names that start with `observe_` , and match the `callRemote` invocations from the sender side just as the usual `remote_*` and `perspective_*` methods match normal `callRemote` calls.

The first time a reference to the `pb.Cacheable` object is sent to any particular recipient, a sender-side Observer will be created for it, and the `getStateToCacheAndObserveFor` method will be called to get the current state and register the Observer. The state which that returns is sent to the remote end and turned into a local representation using `setCopyableState` just like `pb.RemoteCopy` , described above (in fact it inherits from that class).

After that, your "setter" functions on the sender side should call `callRemote` on the Observer, which causes `observe_*` methods to run on the receiver, which are then supposed to update the receiver-local (cached) state.

When the receiver stops following the cached object and the last reference goes away, the `pb.RemoteCache` object can be freed. Just before it dies, it tells the sender side it no longer cares about the original object. When *that* reference count goes to zero, the Observer goes away and the `pb.Cacheable` object can stop announcing every change that takes place. The stoppedObserving method is used to tell the `pb.Cacheable` that the Observer has gone away.

With the `pb.Cacheable` and `pb.RemoteCache` classes in place, bound together by a call to `pb.setUnjellyableForClass` , all that remains is to pass a reference to your `pb.Cacheable` over the wire to the remote end. The corresponding `pb.RemoteCache` object will automatically be created, and the matching methods will be used to keep the receiver-side slave object in sync with the sender-side master object.

Example

Here is a complete example, in which the `MasterDuckPond` is controlled by the sending side, and the `SlaveDuckPond` is a cache that tracks changes to the master:

`cache_classes.py`

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from twisted.spread import pb

class MasterDuckPond(pb.Cacheable):
    def __init__(self, ducks):
        self.observers = []
        self.ducks = ducks
    def count(self):
        print("I have [%d] ducks" % len(self.ducks))
    def addDuck(self, duck):
        self.ducks.append(duck)
        for o in self.observers: o.callRemote('addDuck', duck)
```

[4] This is actually a RemoteCacheObserver , but it isn't very useful to subclass or modify, so simply treat it as a little demon that sits in your `pb.Cacheable` class and helps you distribute change notifications. The only useful thing to do with it is to run its `callRemote` method, which acts just like a normal `pb.Referenceable` 's method of the same name.

```
    def removeDuck(self, duck):
        self.ducks.remove(duck)
        for o in self.observers: o.callRemote('removeDuck', duck)
    def getStateToCacheAndObserveFor(self, perspective, observer):
        self.observers.append(observer)
        # you should ignore pb.Cacheable-specific state, like self.observers
        return self.ducks # in this case, just a list of ducks
    def stoppedObserving(self, perspective, observer):
        self.observers.remove(observer)

class SlaveDuckPond(pb.RemoteCache):
    # This is a cache of a remote MasterDuckPond
    def count(self):
        return len(self.cacheducks)
    def getDucks(self):
        return self.cacheducks
    def setCopyableState(self, state):
        print(" cache - sitting, er, setting ducks")
        self.cacheducks = state
    def observe_addDuck(self, newDuck):
        print(" cache - addDuck")
        self.cacheducks.append(newDuck)
    def observe_removeDuck(self, deadDuck):
        print(" cache - removeDuck")
        self.cacheducks.remove(deadDuck)

pb.setUnjellyableForClass(MasterDuckPond, SlaveDuckPond)
```

cache_sender.py

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from twisted.spread import pb, jelly
from twisted.python import log
from twisted.internet import reactor
from cache_classes import MasterDuckPond

class Sender:
    def __init__(self, pond):
        self.pond = pond

    def phase1(self, remote):
        self.remote = remote
        d = remote.callRemote("takePond", self.pond)
        d.addCallback(self.phase2).addErrback(log.err)
    def phase2(self, response):
        self.pond.addDuck("ugly duckling")
        self.pond.count()
        reactor.callLater(1, self.phase3)
    def phase3(self):
        d = self.remote.callRemote("checkDucks")
        d.addCallback(self.phase4).addErrback(log.err)
    def phase4(self, dummy):
        self.pond.removeDuck("one duck")
        self.pond.count()
```

```
        self.remote.callRemote("checkDucks")
        d = self.remote.callRemote("ignorePond")
        d.addCallback(self.phase5)
    def phase5(self, dummy):
        d = self.remote.callRemote("shutdown")
        d.addCallback(self.phase6)
    def phase6(self, dummy):
        reactor.stop()

def main():
    master = MasterDuckPond(["one duck", "two duck"])
    master.count()

    sender = Sender(master)
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 8800, factory)
    deferred = factory.getRootObject()
    deferred.addCallback(sender.phase1)
    reactor.run()

if __name__ == '__main__':
    main()
```

cache_receiver.py

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from twisted.application import service, internet
from twisted.internet import reactor
from twisted.spread import pb
import cache_classes

class Receiver(pb.Root):
    def remote_takePond(self, pond):
        self.pond = pond
        print("got pond:", pond) # a DuckPondCache
        self.remote_checkDucks()
    def remote_checkDucks(self):
        print("[%d] ducks: " % self.pond.count(), self.pond.getDucks())
    def remote_ignorePond(self):
        # stop watching the pond
        print("dropping pond")
        # gc causes __del__ causes 'decache' msg causes stoppedObserving
        self.pond = None
    def remote_shutdown(self):
        reactor.stop()

application = service.Application("copy_receiver")
internet.TCPServer(8800, pb.PBServerFactory(Receiver())).setServiceParent(
    service.IServiceCollection(application))
```

When run, this example emits the following:

```
$ twistd -n -y cache_receiver.py
[-] twisted.spread.pb.PBServerFactory starting on 8800
[-] Starting factory <twisted.spread.pb.PBServerFactory instance at
0x40615acc>
[Broker,0,127.0.0.1]  cache - sitting, er, setting ducks
[Broker,0,127.0.0.1] got pond: <cache_classes.SlaveDuckPond instance at
0x406eb5ec>
[Broker,0,127.0.0.1] [2] ducks:  ['one duck', 'two duck']
[Broker,0,127.0.0.1]  cache - addDuck
[Broker,0,127.0.0.1] [3] ducks:  ['one duck', 'two duck', 'ugly duckling']
[Broker,0,127.0.0.1]  cache - removeDuck
[Broker,0,127.0.0.1] [2] ducks:  ['two duck', 'ugly duckling']
[Broker,0,127.0.0.1] dropping pond
```

```
$ ./cache_sender.py
I have [2] ducks
I have [3] ducks
I have [2] ducks
Main loop terminated.
```

Points to notice:

- There is one `Observer` for each remote program that holds an active reference. Multiple references inside the same program don't matter: the serialization layer notices the duplicates and does the appropriate reference counting[5] .
- Multiple Observers need to be kept in a list, and all of them need to be updated when something changes. By sending the initial state at the same time as you add the observer to the list, in a single atomic action that cannot be interrupted by a state change, you insure that you can send the same status update to all the observers.
- The `observer.callRemote` calls can still fail. If the remote side has disconnected very recently and `stoppedObserving` has not yet been called, you may get a `DeadReferenceError` . It is a good idea to add an errback to those `callRemote` s to throw away such an error. This is a useful idiom:

```
observer.callRemote('foo', arg).addErrback(lambda f: None)
```

- `getStateToCacheAndObserverFor` must return some object that represents the current state of the object. This may simply be the object's `__dict__` attribute. It is a good idea to remove the `pb.Cacheable` -specific members of it before sending it to the remote end. The list of Observers, in particular, should be left out, to avoid dizzying recursive Cacheable references. The mind boggles as to the potential consequences of leaving in such an item.
- A `perspective` argument is available to `getStateToCacheAndObserveFor` , as well as `stoppedObserving` . I think the purpose of this is to allow viewer-specific changes to the way the cache is updated. If all remote viewers are supposed to see the same data, it can be ignored.

More Information

- The best source for information comes from the docstrings in twisted.spread.flavors , where `pb.Cacheable` is implemented.
- The spread.publish module also uses `Cacheable` , and might be a source of further information.

[5] This applies to multiple references through the same Broker . If you've managed to make multiple TCP connections to the same program, you deserve whatever you get.

Authentication with Perspective Broker

Overview

The examples shown in *Using Perspective Broker* demonstrate how to do basic remote method calls, but provided no facilities for authentication. In this context, authentication is about who gets which remote references, and how to restrict access to the "right" set of people or programs.

As soon as you have a program which offers services to multiple users, where those users should not be allowed to interfere with each other, you need to think about authentication. Many services use the idea of an "account" , and rely upon fact that each user has access to only one account. Twisted uses a system called *cred* to handle authentication issues, and Perspective Broker has code to make it easy to implement the most common use cases.

Compartmentalizing Services

Imagine how you would write a chat server using PB. The first step might be a `ChatServer` object which had a bunch of `pb.RemoteReference` s that point at user clients. Pretend that those clients offered a `remote_print` method which lets the server print a message on the user's console. In that case, the server might look something like this:

```
class ChatServer(pb.Referenceable):

    def __init__(self):
        self.groups = {} # indexed by name
        self.users = {} # indexed by name
    def remote_joinGroup(self, username, groupname):
        if groupname not in self.groups:
            self.groups[groupname] = []
        self.groups[groupname].append(self.users[username])
    def remote_sendMessage(self, from_username, groupname, message):
        group = self.groups[groupname]
        if group:
            # send the message to all members of the group
            for user in group:
                user.callRemote("print",
                                "<%s> says: %s" % (from_username,
                                                   message))
```

For now, assume that all clients have somehow acquired a `pb.RemoteReference` to this `ChatServer` object, perhaps using `pb.Root` and `getRootObject` as described in the *previous chapter* . In this scheme, when a user sends a message to the group, their client runs something like the following:

```
remotegroup.callRemote("sendMessage", "alice", "Hi, my name is alice.")
```

Incorrect Arguments

You've probably seen the first problem: users can trivially spoof each other. We depend upon the user to pass a correct value in their "username" argument, and have no way to tell if they're lying or not. There is nothing to prevent Alice from modifying her client to do:

```
remotegroup.callRemote("sendMessage", "bob", "i like pork")
```

much to the horror of Bob's vegetarian friends.[1]

(In general, learn to get suspicious if you see any argument of a remotely-invokable method described as "must be X")

The best way to fix this is to keep track of the user's name locally, rather than asking them to send it to the server with each message. The best place to keep state is in an object, so this suggests we need a per-user object. Rather than choosing an obvious name[2] , let's call this the `User` class.

```
class User(pb.Referenceable):
    def __init__(self, username, server, clientref):
        self.name = username
        self.server = server
        self.remote = clientref
    def remote_joinGroup(self, groupname):
        self.server.joinGroup(groupname, self)
    def remote_sendMessage(self, groupname, message):
        self.server.sendMessage(self.name, groupname, message)
    def send(self, message):
        self.remote.callRemote("print", message)

class ChatServer:
    def __init__(self):
        self.groups = {} # indexed by name
    def joinGroup(self, groupname, user):
        if groupname not in self.groups:
            self.groups[groupname] = []
        self.groups[groupname].append(user)
    def sendMessage(self, from_username, groupname, message):
        group = self.groups[groupname]
        if group:
            # send the message to all members of the group
            for user in group:
                user.send("<%s> says: %s" % (from_username, message))
```

Again, assume that each remote client gets access to a single `User` object, which is created with the proper username.

Note how the `ChatServer` object has no remote access: it isn't even `pb.Referenceable` anymore. This means that all access to it must be mediated through other objects, with code that is under your control.

As long as Alice only has access to her own `User` object, she can no longer spoof Bob. The only way for her to invoke `ChatServer.sendMessage` is to call her `User` object's `remote_sendMessage` method, and that method uses its own state to provide the `from_username` argument. It doesn't give her any way to change that state.

This restriction is important. The `User` object is able to maintain its own integrity because there is a wall between the object and the client: the client cannot inspect or modify internal state, like the `.name` attribute. The only way through this wall is via remote method invocations, and the only control Alice has over those invocations is when they get invoked and what arguments they are given.

Note: No object can maintain its integrity against local threats: by design, Python offers no mechanism for class instances to hide their attributes, and once an intruder has a copy of `self.__dict__` , they can do everything the original object was able to do.

---

[1] Apparently Alice is one of those weirdos who has nothing better to do than to try and impersonate Bob. She will lie to her chat client, send incorrect objects to remote methods, even rewrite her local client code entirely to accomplish this juvenile prank. Given this adversarial relationship, one must wonder why she and Bob seem to spend so much time together: their adventures are clearly documented by the cryptographic literature.

[2] The obvious name is clearly `ServerSidePerUserObjectWhichNobodyElseHasAccessTo` , but because Python makes everything else so easy to read, it only seems fair to make your audience work for *something* .

Unforgeable References

Now suppose you wanted to implement group parameters, for example a mode in which nobody was allowed to talk about mattresses because some users were sensitive and calming them down after someone said "mattress" is a hassle that's best avoided altogether. Again, per-group state implies a per-group object. We'll go out on a limb and call this the `Group` object:

```
class User(pb.Referenceable):
    def __init__(self, username, server, clientref):
        self.name = username
        self.server = server
        self.remote = clientref
    def remote_joinGroup(self, groupname, allowMattress=True):
        return self.server.joinGroup(groupname, self, allowMattress)
    def send(self, message):
        self.remote.callRemote("print", message)

class Group(pb.Referenceable):
    def __init__(self, groupname, allowMattress):
        self.name = groupname
        self.allowMattress = allowMattress
        self.users = []
    def remote_send(self, from_user, message):
        if not self.allowMattress and "mattress" in message:
            raise ValueError("Don't say that word")
        for user in self.users:
            user.send("<%s> says: %s" % (from_user.name, message))
    def addUser(self, user):
        self.users.append(user)

class ChatServer:
    def __init__(self):
        self.groups = {} # indexed by name
    def joinGroup(self, groupname, user, allowMattress):
        if groupname not in self.groups:
            self.groups[groupname] = Group(groupname, allowMattress)
        self.groups[groupname].addUser(user)
        return self.groups[groupname]
```

This example takes advantage of the fact that `pb.Referenceable` objects sent over a wire can be returned to you, and they will be turned into references to the same object that you originally sent. The client cannot modify the object in any way: all they can do is point at it and invoke its `remote_*` methods. Thus, you can be sure that the `.name` attribute remains the same as you left it. In this case, the client code would look something like this:

```
class ClientThing(pb.Referenceable):
    def remote_print(self, message):
        print(message)
    def join(self):
        d = self.remoteUser.callRemote("joinGroup", "#twisted",
                                       allowMattress=False)
        d.addCallback(self.gotGroup)
    def gotGroup(self, group):
        group.callRemote("send", self.remoteUser, "hi everybody")
```

The `User` object is sent from the server side, and is turned into a `pb.RemoteReference` when it arrives at the client. The client sends it back to `Group.remote_send`, and PB turns it back into a reference to the original `User` when it gets there. `Group.remote_send` can then use its `.name` attribute as the sender of the message.

Note: Third party references (there aren't any)

This technique also relies upon the fact that the `pb.Referenceable` reference can *only* come from someone who holds a corresponding `pb.RemoteReference` . The design of the serialization mechanism (implemented in twisted.spread.jelly : pb, jelly, spread.. get it? Look for "banana" , too. What other networking framework can claim API names based on sandwich ingredients?) makes it impossible for a client to obtain a reference that they weren't explicitly given. References passed over the wire are given id numbers and recorded in a per-connection dictionary. If you didn't give them the reference, the id number won't be in the dict, and no amount of guessing by a malicious client will give them anything else. The dict goes away when the connection is dropped, further limiting the scope of those references.

Furthermore, it is not possible for Bob to send *his* `User` reference to Alice (perhaps over some other PB channel just between the two of them). Outside the context of Bob's connection to the server, that reference is just a meaningless number. To prevent confusion, PB will tell you if you try to give it away: when you try to hand a `pb.RemoteReference` to a third party, you'll get an exception (implemented with an assert in pb.py:364 RemoteReference.jellyFor).

This helps the security model somewhat: only the client you gave the reference to can cause any damage with it. Of course, the client might be a brainless zombie, simply doing anything some third party wants. When it's not proxying `callRemote` invocations, it's probably terrorizing the living and searching out human brains for sustenance. In short, if you don't trust them, don't give them that reference.

And remember that everything you've ever given them over that connection can come back to you. If expect the client to invoke your method with some object A that you sent to them earlier, and instead they send you object B (that you also sent to them earlier), and you don't check it somehow, then you've just opened up a security hole (we'll see an example of this shortly). It may be better to keep such objects in a dictionary on the server side, and have the client send you an index string instead. Doing it that way makes it obvious that they can send you anything they want, and improves the chances that you'll remember to implement the right checks. (This is exactly what PB is doing underneath, with a per-connection dictionary of `Referenceable` objects, indexed by a number).

And, of course, you have to make sure you don't accidentally hand out a reference to the wrong object.

But again, note the vulnerability. If Alice holds a `RemoteReference` to *any* object on the server side that has a `.name` attribute, she can use that name as a spoofed"from" parameter. As a simple example, what if her client code looked like:

```
class ClientThing(pb.Referenceable):
    def join(self):
        d = self.remoteUser.callRemote("joinGroup", "#twisted")
        d.addCallback(self.gotGroup)
    def gotGroup(self, group):
        group.callRemote("send", from_user=group, "hi everybody")
```

This would let her send a message that appeared to come from "#twisted" rather than "Alice" . If she joined a group that happened to be named "bob" (perhaps it is the "How To Be Bob" channel, populated by Alice and countless others, a place where they can share stories about their best impersonating-Bob moments), then she would be able to emit a message that looked like "<bob> says: hi there" , and she has accomplished her lifelong goal.

Argument Typechecking

There are two techniques to close this hole. The first is to have your remotely-invokable methods do type-checking on their arguments: if `Group.remote_send` asserted `isinstance(from_user, User)` then Alice couldn't use non-User objects to do her spoofing, and hopefully the rest of the system is designed well enough to prevent her from obtaining access to somebody else's User object.

Objects as Capabilities

The second technique is to avoid having the client send you the objects altogether. If they don't send you anything, there is nothing to verify. In this case, you would have to have a per-user-per-group object, in which the `remote_send` method would only take a single `message` argument. The `UserGroup` object is created with references to the only `User` and `Group` objects that it will ever use, so no lookups are needed:

```
class UserGroup(pb.Referenceable):
    def __init__(self, user, group):
        self.user = user
        self.group = group
    def remote_send(self, message):
        self.group.send(self.user.name, message)

class Group:
    def __init__(self, groupname, allowMattress):
        self.name = groupname
        self.allowMattress = allowMattress
        self.users = []
    def send(self, from_user, message):
        if not self.allowMattress and "mattress" in message:
            raise ValueError("Don't say that word")
        for user in self.users:
            user.send("<%s> says: %s" % (from_user.name, message))
    def addUser(self, user):
        self.users.append(user)
```

The only message-sending method Alice has left is `UserGroup.remote_send` , and it only accepts a message: there are no remaining ways to influence the "from" name.

In this model, each remotely-accessible object represents a very small set of capabilities. Security is achieved by only granting a minimal set of abilities to each remote user.

PB provides a shortcut which makes this technique easier to use. The `Viewable` class will be discussed *below* .

Avatars and Perspectives

In Twisted's *cred* system, an "Avatar" is an object that lives on the "server" side (defined here as the side farthest from the human who is trying to get something done) which lets the remote user get something done. The avatar isn't really a particular class, it's more like a description of a role that some object plays, as in "the Foo object here is acting as the user's avatar for this particular service" . Generally, the remote user has some way of getting their avatar to run some code. The avatar object may enforce some security checks, and provide additional data, then call other methods which get things done.

The two pieces in the cred puzzle (for any protocol, not just PB) are: "what serves as the Avatar?" , and "how does the user get access to it?" .

For PB, the first question is easy. The Avatar is a remotely-accessible object which can run code: this is a perfect description of `pb.Referenceable` and its subclasses. We shall defer the second question until the next section.

In the example above, you can think of the `ChatServer` and `Group` objects as a service. The `User` object is the user's server-side representative: everything the user is capable of doing is done by running one of its methods. Anything that the server wants to do to the user (change their group membership, change their name, delete their pet cat, whatever) is done by manipulating the `User` object.

There are multiple User objects living in peace and harmony around the ChatServer. Each has a different point of view on the services provided by the ChatServer and the Groups: each may belong to different groups, some might have more permissions than others (like the ability to create groups). These different points of view are called "Perspectives"

. This is the origin of the term "Perspective" in "Perspective Broker" : PB provides and controls (i.e. "brokers") access to Perspectives.

Once upon a time, these local-representative objects were actually called `pb.Perspective` . But this has changed with the advent of the rewritten cred system, and now the more generic term for a local representative object is an Avatar. But you will still see reference to "Perspective" in the code, the docs, and the module names[3] . Just remember that perspectives and avatars are basically the same thing.

Despite all we've been *telling you* about how Avatars are more of a concept than an actual class, the base class from which you can create your server-side avatar-ish objects is, in fact, named `pb.Avatar`[4] . These objects behave very much like `pb.Referenceable` . The only difference is that instead of offering "remote_FOO" methods, they offer "perspective_FOO" methods.

The other way in which `pb.Avatar` differs from `pb.Referenceable` is that the avatar objects are designed to be the first thing retrieved by a cred-using remote client. Just as `PBClientFactory.getRootObject` gives the client access to a `pb.Root` object (which can then provide access to all kinds of other objects), `PBClientFactory.login` gives client access to a `pb.Avatar` object (which can return other references).

So, the first half of using cred in your PB application is to create an Avatar object which implements `perspective_` methods and is careful to do useful things for the remote user while remaining vigilant against being tricked with unexpected argument values. It must also be careful to never give access to objects that the user should not have access to, whether by returning them directly, returning objects which contain them, or returning objects which can be asked (remotely) to provide them.

The second half is how the user gets a `pb.RemoteReference` to your Avatar. As explained *elsewhere* , Avatars are obtained from a Realm. The Realm doesn't deal with authentication at all (usernames, passwords, public keys, challenge-response systems, retinal scanners, real-time DNA sequencers, etc). It simply takes an "avatarID" (which is effectively a username) and returns an Avatar object. The Portal and its Checkers deal with authenticating the user: by the time they are done, the remote user has proved their right to access the avatarID that is given to the Realm, so the Realm can return a remotely-controllable object that has whatever powers you wish to grant to this particular user.

For PB, the realm is expected to return a `pb.Avatar` (or anything which implements `pb.IPerspective` , really, but there's no reason to not return a `pb.Avatar` subclass). This object will be given to the client just like a `pb.Root` would be without cred, and the user can get access to other objects through it (if you let them).

The basic idea is that there is a separate IPerspective-implementing object (i.e. the Avatar subclass) (i.e. the "perspective") for each user, and *only* the authorized user gets a remote reference to that object. You can store whatever permissions or capabilities the user possesses in that object, and then use them when the user invokes a remote method. You give the user access to the perspective object instead of the objects that do the real work.

Perspective Examples

Here is a brief example of using a pb.Avatar. Most of the support code is magic for now: we'll explain it later.

One Client

`pb5server.py`

[3] We could just go ahead and rename Perspective Broker to be Avatar Broker, but 1) that would cause massive compatibility problems, and 2) "AB" doesn't fit into the whole sandwich-themed naming scheme nearly as well as "PB" does. If we changed it to AB, we'd probably have to change Banana to be CD (CoderDecoder), and Jelly to be EF (EncapsulatorFragmentor). twisted.spread would then have to be renamed twisted.alphabetsoup, and then the whole food-pun thing would start all over again.

[4] The avatar-ish class is named `pb.Avatar` because `pb.Perspective` was already taken, by the (now obsolete) oldcred perspective-ish class. It is a pity, but it simply wasn't possible both replace `pb.Perspective` in-place *and* maintain a reasonable level of backwards-compatibility.

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from zope.interface import implementer

from twisted.spread import pb
from twisted.cred import checkers, portal
from twisted.internet import reactor

class MyPerspective(pb.Avatar):
    def __init__(self, name):
        self.name = name
    def perspective_foo(self, arg):
        print("I am", self.name, "perspective_foo(",arg,") called on", self)

@implementer(portal.IRealm)
class MyRealm:
    def requestAvatar(self, avatarId, mind, *interfaces):
        if pb.IPerspective not in interfaces:
            raise NotImplementedError
        return pb.IPerspective, MyPerspective(avatarId), lambda:None

p = portal.Portal(MyRealm())
p.registerChecker(
    checkers.InMemoryUsernamePasswordDatabaseDontUse(user1="pass1"))
reactor.listenTCP(8800, pb.PBServerFactory(p))
reactor.run()
```

pb5client.py

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from twisted.spread import pb
from twisted.internet import reactor
from twisted.cred import credentials

def main():
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 8800, factory)
    def1 = factory.login(credentials.UsernamePassword("user1", "pass1"))
    def1.addCallback(connected)
    reactor.run()

def connected(perspective):
    print("got perspective ref:", perspective)
    print("asking it to foo(12)")
    perspective.callRemote("foo", 12)

main()
```

Ok, so that wasn't really very exciting. It doesn't accomplish much more than the first PB example, and used a lot more code to do it. Let's try it again with two users this time.

Note: When the client runs `login` to request the Perspective, they can provide it with an optional `client` argument (which must be a `pb.Referenceable` object). If they do, then a reference to that object will be handed to the realm's `requestAvatar` in the `mind` argument.

The server-side Perspective can use it to invoke remote methods on something in the client, so that the client doesn't always have to drive the interaction. In a chat server, the client object would be the one to which"display text" messages were sent. In a board game server, this would provide a way to tell the clients that someone has made a move, so they can update their game boards.

Two Clients

pb6server.py

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from zope.interface import implementer

from twisted.spread import pb
from twisted.cred import checkers, portal
from twisted.internet import reactor

class MyPerspective(pb.Avatar):
    def __init__(self, name):
        self.name = name
    def perspective_foo(self, arg):
        print("I am", self.name, "perspective_foo(",arg,") called on", self)

@implementer(portal.IRealm)
class MyRealm:
    def requestAvatar(self, avatarId, mind, *interfaces):
        if pb.IPerspective not in interfaces:
            raise NotImplementedError
        return pb.IPerspective, MyPerspective(avatarId), lambda:None

p = portal.Portal(MyRealm())
c = checkers.InMemoryUsernamePasswordDatabaseDontUse(user1="pass1",
                                                     user2="pass2")
p.registerChecker(c)
reactor.listenTCP(8800, pb.PBServerFactory(p))
reactor.run()
```

pb6client1.py

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.
```

```
from __future__ import print_function

from twisted.spread import pb
from twisted.internet import reactor
from twisted.cred import credentials

def main():
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 8800, factory)
    def1 = factory.login(credentials.UsernamePassword("user1", "pass1"))
    def1.addCallback(connected)
    reactor.run()

def connected(perspective):
    print("got perspective1 ref:", perspective)
    print("asking it to foo(13)")
    perspective.callRemote("foo", 13)

main()
```

pb6client2.py

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from twisted.spread import pb
from twisted.internet import reactor

from twisted.spread import pb
from twisted.internet import reactor
from twisted.cred import credentials

def main():
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 8800, factory)
    def1 = factory.login(credentials.UsernamePassword("user2", "pass2"))
    def1.addCallback(connected)
    reactor.run()

def connected(perspective):
    print("got perspective2 ref:", perspective)
    print("asking it to foo(14)")
    perspective.callRemote("foo", 14)

main()
```

While pb6server.py is running, try starting pb6client1, then pb6client2. Compare the argument passed by the `.callRemote()` in each client. You can see how each client gets connected to a different Perspective.

How that example worked

Let's walk through the previous example and see what was going on.

First, we created a subclass called `MyPerspective` which is our server-side Avatar. It implements a `perspective_foo` method that is exposed to the remote client.

Second, we created a realm (an object which implements `IRealm`, and therefore implements `requestAvatar`). This realm manufactures `MyPerspective` objects. It makes as many as we want, and names each one with the avatarID (a username) that comes out of the checkers. This MyRealm object returns two other objects as well, which we will describe later.

Third, we created a portal to hold this realm. The portal's job is to dispatch incoming clients to the credential checkers, and then to request Avatars for any which survive the authentication process.

Fourth, we made a simple checker (an object which implements `IChecker`) to hold valid user/password pairs. The checker gets registered with the portal, so it knows who to ask when new clients connect. We use a checker named `InMemoryUsernamePasswordDatabaseDontUse`, which suggests that 1: all the username/password pairs are kept in memory instead of being saved to a database or something, and 2: you shouldn't use it. The admonition against using it is because there are better schemes: keeping everything in memory will not work when you have thousands or millions of users to keep track of, the passwords will be stored in the .tap file when the application shuts down (possibly a security risk), and finally it is a nuisance to add or remove users after the checker is constructed.

Fifth, we create a `pb.PBServerFactory` to listen on a TCP port. This factory knows how to connect the remote client to the Portal, so incoming connections will be handed to the authentication process. Other protocols (non-PB) would do something similar: the factory that creates Protocol objects will give those objects access to the Portal so authentication can take place.

On the client side, a `pb.PBClientFactory` is created (as *before*) and attached to a TCP connection. When the connection completes, the factory will be asked to produce a Protocol, and it will create a PB object. Unlike the previous chapter, where we used `.getRootObject`, here we use `factory.login` to initiate the cred authentication process. We provide a `credentials` object, which is the client-side agent for doing our half of the authentication process. This process may involve several messages: challenges, responses, encrypted passwords, secure hashes, etc. We give our credentials object everything it will need to respond correctly (in this case, a username and password, but you could write a credential that used public-key encryption or even fancier techniques).

`login` returns a Deferred which, when it fires, will return a `pb.RemoteReference` to the remote avatar. We can then do `callRemote` to invoke a `perspective_foo` method on that Avatar.

Anonymous Clients

`pbAnonServer.py`

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

"""
Implement the realm for and run on port 8800 a PB service which allows both
anonymous and username/password based access.

Successful username/password-based login requests given an instance of
MyPerspective with a name which matches the username with which they
authenticated.  Success anonymous login requests are given an instance of
MyPerspective with the name "Anonymous".
"""

from __future__ import print_function

from sys import stdout
```

```
from zope.interface import implementer

from twisted.python.log import startLogging
from twisted.cred.checkers import ANONYMOUS, AllowAnonymousAccess
from twisted.cred.checkers import InMemoryUsernamePasswordDatabaseDontUse
from twisted.cred.portal import IRealm, Portal
from twisted.internet import reactor
from twisted.spread.pb import Avatar, IPerspective, PBServerFactory


class MyPerspective(Avatar):
    """
    Trivial avatar exposing a single remote method for demonstrative
    purposes.  All successful login attempts in this example will result in
    an avatar which is an instance of this class.

    @type name: C{str}
    @ivar name: The username which was used during login or C{"Anonymous"}
    if the login was anonymous (a real service might want to avoid the
    collision this introduces between anonoymous users and authenticated
    users named "Anonymous").
    """
    def __init__(self, name):
        self.name = name


    def perspective_foo(self, arg):
        """
        Print a simple message which gives the argument this method was
        called with and this avatar's name.
        """
        print("I am %s.  perspective_foo(%s) called on %s." % (
            self.name, arg, self))



@implementer(IRealm)
class MyRealm(object):
    """
    Trivial realm which supports anonymous and named users by creating
    avatars which are instances of MyPerspective for either.
    """
    def requestAvatar(self, avatarId, mind, *interfaces):
        if IPerspective not in interfaces:
            raise NotImplementedError("MyRealm only handles IPerspective")
        if avatarId is ANONYMOUS:
            avatarId = "Anonymous"
        return IPerspective, MyPerspective(avatarId), lambda: None



def main():
    """
    Create a PB server using MyRealm and run it on port 8800.
    """
    startLogging(stdout)
```

```
    p = Portal(MyRealm())

    # Here the username/password checker is registered.
    c1 = InMemoryUsernamePasswordDatabaseDontUse(user1="pass1", user2="pass2")
    p.registerChecker(c1)

    # Here the anonymous checker is registered.
    c2 = AllowAnonymousAccess()
    p.registerChecker(c2)

    reactor.listenTCP(8800, PBServerFactory(p))
    reactor.run()


if __name__ == '__main__':
    main()
```

pbAnonClient.py

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

"""
Client which will talk to the server run by pbAnonServer.py, logging in
either anonymously or with username/password credentials.
"""

from __future__ import print_function

from sys import stdout

from twisted.python.log import err, startLogging
from twisted.cred.credentials import Anonymous, UsernamePassword
from twisted.internet import reactor
from twisted.internet.defer import gatherResults
from twisted.spread.pb import PBClientFactory


def error(why, msg):
    """
    Catch-all errback which simply logs the failure.  This isn't expected to
    be invoked in the normal case for this example.
    """
    err(why, msg)


def connected(perspective):
    """
    Login callback which invokes the remote "foo" method on the perspective
    which the server returned.
    """
    print("got perspective1 ref:", perspective)
    print("asking it to foo(13)")
    return perspective.callRemote("foo", 13)
```

```
def finished(ignored):
    """
    Callback invoked when both logins and method calls have finished to shut
    down the reactor so the example exits.
    """
    reactor.stop()


def main():
    """
    Connect to a PB server running on port 8800 on localhost and log in to
    it, both anonymously and using a username/password it will recognize.
    """
    startLogging(stdout)
    factory = PBClientFactory()
    reactor.connectTCP("localhost", 8800, factory)

    anonymousLogin = factory.login(Anonymous())
    anonymousLogin.addCallback(connected)
    anonymousLogin.addErrback(error, "Anonymous login failed")

    usernameLogin = factory.login(UsernamePassword("user1", "pass1"))
    usernameLogin.addCallback(connected)
    usernameLogin.addErrback(error, "Username/password login failed")

    bothDeferreds = gatherResults([anonymousLogin, usernameLogin])
    bothDeferreds.addCallback(finished)

    reactor.run()


if __name__ == '__main__':
    main()
```

pbAnonServer.py implements a server based on pb6server.py, extending it to permit anonymous logins in addition to authenticated logins. An AllowAnonymousAccess checker and an InMemoryUsernamePasswordDatabaseDontUse checker are registered and the client's choice of credentials object determines which is used to authenticate the login. In either case, the realm will be called on to create an avatar for the login. `AllowAnonymousAccess` always produces an `avatarId` of `twisted.cred.checkers.ANONYMOUS` .

On the client side, the only change is the use of an instance of Anonymous when calling PBClientFactory.login .

Using Avatars

Avatar Interfaces

The first element of the 3-tuple returned by `requestAvatar` indicates which Interface this Avatar implements. For PB avatars, it will always be `pb.IPerspective` , because that's the only interface these avatars implement.

This element is present because `requestAvatar` is actually presented with a list of possible Interfaces. The question being posed to the Realm is: "do you have an avatar for (avatarID) that can implement one of the following set of Interfaces?" . Some portals and checkers might give a list of Interfaces and the Realm could pick; the PB code only knows how to do one, so we cannot take advantage of this feature.

Logging Out

The third element of the 3-tuple is a zero-argument callable, which will be invoked by the protocol when the connection has been lost. We can use this to notify the Avatar when the client has lost its connection. This will be described in more detail below.

Making Avatars

In the example above, we create Avatars upon request, during `requestAvatar` . Depending upon the service, these Avatars might already exist before the connection is received, and might outlive the connection. The Avatars might also accept multiple connections.

Another possibility is that the Avatars might exist ahead of time, but in a different form (frozen in a pickle and/or saved in a database). In this case, `requestAvatar` may need to perform a database lookup and then do something with the result before it can provide an avatar. In this case, it would probably return a Deferred so it could provide the real Avatar later, once the lookup had completed.

Here are some possible implementations of `MyRealm.requestAvatar` :

```
# pre-existing, static avatars
def requestAvatar(self, avatarID, mind, *interfaces):
    assert pb.IPerspective in interfaces
    avatar = self.avatars[avatarID]
    return pb.IPerspective, avatar, lambda:None

# database lookup and unpickling
def requestAvatar(self, avatarID, mind, *interfaces):
    assert pb.IPerspective in interfaces
    d = self.database.fetchAvatar(avatarID)
    d.addCallback(self.doUnpickle)
    return pb.IPerspective, d, lambda:None
def doUnpickle(self, pickled):
    avatar = pickle.loads(pickled)
    return avatar

# everybody shares the same Avatar
def requestAvatar(self, avatarID, mind, *interfaces):
    assert pb.IPerspective in interfaces
    return pb.IPerspective, self.theOneAvatar, lambda:None

# anonymous users share one Avatar, named users each get their own
def requestAvatar(self, avatarID, mind, *interfaces):
    assert pb.IPerspective in interfaces
    if avatarID == checkers.ANONYMOUS:
        return pb.IPerspective, self.anonAvatar, lambda:None
    else:
        return pb.IPerspective, self.avatars[avatarID], lambda:None

# anonymous users get independent (but temporary) Avatars
# named users get their own persistent one
def requestAvatar(self, avatarID, mind, *interfaces):
    assert pb.IPerspective in interfaces
    if avatarID == checkers.ANONYMOUS:
        return pb.IPerspective, MyAvatar(), lambda:None
    else:
        return pb.IPerspective, self.avatars[avatarID], lambda:None
```

The last example, note that the new `MyAvatar` instance is not saved anywhere: it will vanish when the connection is dropped. By contrast, the avatars that live in the `self.avatars` dictionary will probably get persisted into the .tap file along with the Realm, the Portal, and anything else that is referenced by the top-level Application object. This is an easy way to manage saved user profiles.

Connecting and Disconnecting

It may be useful for your Avatars to be told when remote clients gain (and lose) access to them. For example, and Avatar might be updated by something in the server, and if there are clients attached, it should update them (through the "mind" argument which lets the Avatar do callRemote on the client).

One common idiom which accomplishes this is to have the Realm tell the avatar that a remote client has just attached. The Realm can also ask the protocol to let it know when the connection goes away, so it can then inform the Avatar that the client has detached. The third member of the `requestAvatar` return tuple is a callable which will be invoked when the connection is lost.

```
class MyPerspective(pb.Avatar):
    def __init__(self):
        self.clients = []
    def attached(self, mind):
        self.clients.append(mind)
        print("attached to", mind)
    def detached(self, mind):
        self.clients.remove(mind)
        print("detached from", mind)
    def update(self, message):
        for c in self.clients:
            c.callRemote("update", message)

class MyRealm:
    def requestAvatar(self, avatarID, mind, *interfaces):
        assert pb.IPerspective in interfaces
        avatar = self.avatars[avatarID]
        avatar.attached(mind)
        return pb.IPerspective, avatar, lambda a=avatar:a.detached(mind)
```

Viewable

Once you have `IPerspective` objects (i.e. the Avatar) to represent users, the Viewable class can come into play. This class behaves a lot like `Referenceable` : it turns into a `RemoteReference` when sent over the wire, and certain methods can be invoked by the holder of that reference. However, the methods that can be called have names that start with `view_` instead of `remote_` , and those methods are always called with an extra `perspective` argument that points to the Avatar through which the reference was sent:

```
class Foo(pb.Viewable):
    def view_doFoo(self, perspective, arg1, arg2):
        pass
```

This is useful if you want to let multiple clients share a reference to the same object. The `view_` methods can use the "perspective" argument to figure out which client is calling them. This gives them a way to do additional permission checks, do per-user accounting, etc.

This is the shortcut which makes per-user-per-group capability objects much easier to use. Instead of creating such per-(user,group) objects, you just have per-group objects which inherit from `pb.Viewable` , and give the user references to them. The local `pb.Avatar` object will automatically show up as the "perspective" argument in the `view_*` method calls, give you a chance to involve the Avatar in the process.

Chat Server with Avatars

Combining all the above techniques, here is an example chat server which uses a fixed set of identities (say, for the three members of your bridge club, who hang out in "#NeedAFourth" hoping that someone will discover your server, guess somebody's password, break in, join the group, and also be available for a game next Saturday afternoon).

`chatserver.py`

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from zope.interface import implementer

from twisted.cred import portal, checkers
from twisted.spread import pb
from twisted.internet import reactor

class ChatServer:
    def __init__(self):
        self.groups = {} # indexed by name

    def joinGroup(self, groupname, user, allowMattress):
        if groupname not in self.groups:
            self.groups[groupname] = Group(groupname, allowMattress)
        self.groups[groupname].addUser(user)
        return self.groups[groupname]

@implementer(portal.IRealm)
class ChatRealm:
    def requestAvatar(self, avatarID, mind, *interfaces):
        assert pb.IPerspective in interfaces
        avatar = User(avatarID)
        avatar.server = self.server
        avatar.attached(mind)
        return pb.IPerspective, avatar, lambda a=avatar:a.detached(mind)

class User(pb.Avatar):
    def __init__(self, name):
        self.name = name
    def attached(self, mind):
        self.remote = mind
    def detached(self, mind):
        self.remote = None
    def perspective_joinGroup(self, groupname, allowMattress=True):
        return self.server.joinGroup(groupname, self, allowMattress)
    def send(self, message):
        self.remote.callRemote("print", message)

class Group(pb.Viewable):
    def __init__(self, groupname, allowMattress):
```

```
        self.name = groupname
        self.allowMattress = allowMattress
        self.users = []
    def addUser(self, user):
        self.users.append(user)
    def view_send(self, from_user, message):
        if not self.allowMattress and "mattress" in message:
            raise ValueError("Don't say that word")
        for user in self.users:
            user.send("<%s> says: %s" % (from_user.name, message))

realm = ChatRealm()
realm.server = ChatServer()
checker = checkers.InMemoryUsernamePasswordDatabaseDontUse()
checker.addUser("alice", "1234")
checker.addUser("bob", "secret")
checker.addUser("carol", "fido")
p = portal.Portal(realm, [checker])

reactor.listenTCP(8800, pb.PBServerFactory(p))
reactor.run()
```

Notice that the client uses `perspective_joinGroup` to both join a group and retrieve a `RemoteReference` to the `Group` object. However, the reference they get is actually to a special intermediate object called a `pb.ViewPoint`. When they do `group.callRemote("send", "message")`, their avatar is inserted into the argument list that `Group.view_send` actually sees. This lets the group get their username out of the Avatar without giving the client an opportunity to spoof someone else.

The client side code that joins a group and sends a message would look like this:

`chatclient.py`

```
#!/usr/bin/env python
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

from __future__ import print_function

from twisted.spread import pb
from twisted.internet import reactor
from twisted.cred import credentials

class Client(pb.Referenceable):

    def remote_print(self, message):
        print(message)

    def connect(self):
        factory = pb.PBClientFactory()
        reactor.connectTCP("localhost", 8800, factory)
        def1 = factory.login(credentials.UsernamePassword("alice", "1234"),
                             client=self)
        def1.addCallback(self.connected)
        reactor.run()

    def connected(self, perspective):
        print("connected, joining group #NeedAFourth")
        # this perspective is a reference to our User object.  Save a reference
```

```
        # to it here, otherwise it will get garbage collected after this call,
        # and the server will think we logged out.
        self.perspective = perspective
        d = perspective.callRemote("joinGroup", "#NeedAFourth")
        d.addCallback(self.gotGroup)

    def gotGroup(self, group):
        print("joined group, now sending a message to all members")
        # 'group' is a reference to the Group object (through a ViewPoint)
        d = group.callRemote("send", "You can call me Al.")
        d.addCallback(self.shutdown)

    def shutdown(self, result):
        reactor.stop()


Client().connect()
```

PB Limits

There are a number of limits you might encounter when using Perspective Broker. This document is an attempt to prepare you for as many of them as possible so you can avoid them or at least recognize them when you do run into them.

Banana Limits

Perspective Broker is implemented in terms of a simpler, less functional protocol called Banana. Twisted's implementation of Banana imposes a limit on the length of any sequence-like data type. This applies directly to lists and strings and indirectly to dictionaries, instances and other types. The purpose of this limit is to put an upper bound on the amount of memory which will be allocated to handle a message received over the network. Without, a malicious peer could easily perform a denial of service attack resulting in exhaustion of the receiver's memory. The basic limit is 640 * 1024 bytes, defined by `twisted.spread.banana.SIZE_LIMIT` . It's possible to raise this limit by changing this value (but take care to change it on both sides of the connection).

Another limit imposed by Twisted's Banana implementation is a limit on the size of long integers. The purpose of this limit is the same as the `SIZE_LIMIT` . By default, only integers between -2 ** 448 and 2 ** 448 (exclusive) can be transferred. This limit can be changed using twisted.spread.banana.setPrefixLimit .

Perspective Broker Limits

Perspective Broker imposes an additional limit on top of these lower level limits. The number of local objects for which remote references may exist at a single time over a single connection, by default, is limited to 1024, defined by `twisted.spread.pb.MAX_BROKER_REFS` . This limit also exists to prevent memory exhaustion attacks.

Porting to Python 3

Introduction

Twisted is currently being ported to work with Python 3.3+. This document covers Twisted-specific issues in porting your code to Python 3.

Most, but not all, of Twisted has been ported, and therefore only a subset of modules are installed under Python 3. You can see the remaining modules that need to be ported at twisted.python._setup.notPortedModules, if it is not listed there, then most of all of that module will be ported.

API Differences

twisted.python.failure

Failure.trap raises itself (i.e. a Failure) in Python 2. In Python 3, the wrapped exception will be re-raised.

Byte Strings and Text Strings

Several APIs which on Python 2 accepted or produced byte strings (instances of `str` , sometimes just called *bytes*) have changed to accept or produce text strings (instances of `str` , sometimes just called *text* or *unicode*) on Python 3.

From `twisted.internet.address` , the `IPv4Address` and `IPv6Address` classes have had two attributes change from byte strings to text strings: `type` and `host` .

`twisted.python.log` has shifted significantly towards text strings from byte strings. Logging events, particular those produced by a call like `msg("foo")` , must now be text strings. Consequently, on Python 3, event dictionaries passed to log observes will contain text strings where they previously contained byte strings.

`twisted.python.runtime.platformType` and the return value from `twisted.python.runtime.Platform.getType` are now both text strings.

`twisted.python.filepath.FilePath` has *not* changed. It supports only byte strings. This will probably require applications to update their usage of `FilePath` , at least to pass explicit byte string literals rather than "native" string literals (which are text on Python 3).

`reactor.addSystemEventTrigger` arguments that were previously byte strings are now native strings.

`twisted.names.dns` deals with strings with a wide range of meanings, often several for each DNS record type. Most of these strings have remained as byte strings, which will probably require application updates (for the reason given in the `FilePath` section above). Some strings have changed to text strings, though. Any string representing a human readable address (for example, `Record_A` 's `address` parameter) is now a text string. Additionally, time-to-live (ttl) values given as strings must now be given as text strings.

`twisted.web.resource.IResource` continues to deal with URLs and all URL-derived values as byte strings.

`twisted.web.resource.ErrorPage` has several string attributes (`template` , `brief` , and `detail`) which were previously byte strings. On Python 3 only, these must now be text strings.

Twisted Positioning

`twisted.positioning`: geolocation in Twisted

Introduction

`twisted.positioning` is a package for doing geospatial positioning (trying to find where you are on Earth) using Twisted.

High-level overview

In `twisted.positioning`, you write an IPositioningReceiver implementation that will get called whenever some information about your position is known (such as position, altitude, heading...). The package provides a base class, BasePositioningReceiver you might want to use that implements all of the receiver methods as stubs.

Secondly, you will want a positioning source, which will call your IPositioningReceiver. Currently, `twisted.positioning` provides an NMEA implementation, which is a standard protocol spoken by many positioning devices, usually over a serial port.

Examples

`nmealogger.py`

```
#!/usr/bin/env python
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.
"""
Connects to an NMEA device, logs beacon information and position.
"""
from __future__ import print_function

import sys
from twisted.internet import reactor, serialport
from twisted.positioning import base, nmea
from twisted.python import log, usage


class PositioningReceiver(base.BasePositioningReceiver):
    def positionReceived(self, latitude, longitude):
        log.msg("I'm at {} lat, {} lon".format(latitude, longitude))


    def beaconInformationReceived(self, beaconInformation):
        template = "{0.seen} beacons seen, {0.used} beacons used"
        log.msg(template.format(beaconInformation))



class Options(usage.Options):
    optParameters = [
        ['baud-rate', 'b', 4800, "Baud rate (default: 4800)"],
        ['serial-port', 'p', '/dev/ttyS0', 'Serial Port device'],
    ]



def run():
    log.startLogging(sys.stdout)

    opts = Options()
    try:
        opts.parseOptions()
    except usage.UsageError as message:
        print("{}: {}".format(sys.argv[0], message))
        return
```

```
    positioningReceiver = PositioningReceiver()
    nmeaReceiver = nmea.NMEAAdapter(positioningReceiver)
    proto = nmea.NMEAProtocol(nmeaReceiver)

    port, baudrate = opts["serial-port"], opts["baud-rate"]
    serialport.SerialPort(proto, port, reactor, baudrate=baudrate)

    reactor.run()



if __name__ == "__main__":
    run()
```

- Connects to an NMEA device on a serial port, and reports whenever it receives a position.

Twisted Glossary

adaptee

An object that has been adapted, also called "original" . See *Adapter* .

Adapter

An object whose sole purpose is to implement an Interface for another object. See *Interfaces and Adapters* .

Application

A twisted.application.service.Application . There are HOWTOs on *creating and manipulating* them as a system-administrator, as well as *using* them in your code.

Avatar

(from *Twisted Cred*) business logic for specific user. For example, in *PB* these are perspectives, in POP3 these are mailboxes, and so on.

Banana

The low-level data marshalling layer of *Twisted Spread* . See twisted.spread.banana .

Broker

A twisted.spread.pb.Broker , the object request broker for *Twisted Spread* .

cache

A way to store data in readily accessible place for later reuse. Caching data is often done because the data is expensive to produce or access. Caching data risks being stale, or out of sync with the original data.

component

A special kind of (persistent) Adapter that works with a twisted.python.components.Componentized . See also *Interfaces and Adapters* .

Componentized

A Componentized object is a collection of information, separated into domain-specific or role-specific instances, that all stick together and refer to each other. Each object is an Adapter , which, in the context of Componentized, we call "components" . See also *Interfaces and Adapters* .

conch

Twisted's SSH implementation.

Connector

Object used to interface between client connections and protocols, usually used with a twisted.internet.protocol.ClientFactory to give you control over how a client connection reconnects. See twisted.internet.interfaces.IConnector and *Writing Clients* .

Consumer

An object that consumes data from a *Producer* . See twisted.internet.interfaces.IConsumer .

Cred

Twisted's authentication API, twisted.cred . See *Introduction to Twisted Cred* and *Twisted Cred usage* .

credentials

A username/password, public key, or some other information used for authentication.

credential checker

Where authentication actually happens. See ICredentialsChecker .

CVSToys

A nifty set of tools for CVS, available at http://twistedmatrix.com/users/acapnotic/wares/code/CVSToys/ .

Daemon

A background process that does a job or handles client requests. *Daemon* is a Unix term; *service* is the Windows equivalent.

Deferred

An instance of twisted.internet.defer.Deferred , an abstraction for handling chains of callbacks and error handlers ("errbacks"). See the *Deferring Execution* HOWTO.

Enterprise

Twisted's RDBMS support. It contains twisted.enterprise.adbapi for asynchronous access to any standard DB-API 2.0 module. See *Introduction to Twisted Enterprise* for more details.

errback

A callback attached to a *Deferred* with `.addErrback` to handle errors.

Factory

In general, an object that constructs other objects. In Twisted, a Factory usually refers to a twisted.internet.protocol.Factory , which constructs *Protocol* instances for incoming or outgoing connections. See *Writing Servers* and *Writing Clients* .

Failure

Basically, an asynchronous exception that contains traceback information; these are used for passing errors through asynchronous callbacks.

im

Abbreviation of "(Twisted) *Instance Messenger*" .

Instance Messenger

Instance Messenger is a multi-protocol chat program that comes with Twisted. It can communicate via TOC with the AOL servers, via IRC, as well as via *PB* with *Twisted Words* . See twisted.words.im .

Interface

A class that defines and documents methods that a class conforming to that interface needs to have. A collection of core twisted.internet interfaces can be found in twisted.internet.interfaces . See also *Interfaces and Adapters* .

Jelly

The serialization layer for *Twisted Spread* , although it can be used separately from Twisted Spread as well. It is similar in purpose to Python's standard `pickle` module, but is more network-friendly, and depends on a separate marshaller (*Banana* , in most cases). See twisted.spread.jelly .

Manhole

A debugging/administration interface to a Twisted application.

Microdom

A partial DOM implementation using *SUX* . It is simple and pythonic, rather than strictly standards-compliant. See twisted.web.microdom .

Names

Twisted's DNS server, found in twisted.names .

Nevow

The successor to *Woven* ; available from Divmod .

PB

Abbreviation of "*Perspective Broker*" .

Perspective Broker

The high-level object layer of Twisted *Spread* , implementing semantics for method calling and object copying, caching, and referencing. See twisted.spread.pb .

Portal

Glues *credential checkers* and *realm* s together.

Producer

An object that generates data a chunk at a time, usually to be processed by a *Consumer* . See twisted.internet.interfaces.IProducer .

Protocol

In general each network connection has its own Protocol instance to manage connection-specific state. There is a collection of standard protocol implementations in twisted.protocols . See also *Writing Servers* and *Writing Clients* .

PSU

There is no PSU.

Reactor

The core event-loop of a Twisted application. See *Reactor Basics* .

Reality

See "*Twisted Reality*"

realm

(in *Twisted Cred*) stores *avatars* and perhaps general business logic. See IRealm .

Resource

A twisted.web.resource.Resource , which are served by Twisted Web. Resources can be as simple as a static file on disk, or they can have dynamically generated content.

Service

A twisted.application.service.Service . See *Application howto* for a description of how they relate to *Applications* .

Spread

Twisted Spread is Twisted's remote-object suite. It consists of three layers: *Perspective Broker* , *Jelly* and *Banana.* See *Writing Applications with Perspective Broker* .

SUX

S mall *U* ncomplicated *X* ML, Twisted's simple XML parser written in pure Python. See twisted.web.sux .

TAC

A *T* wisted *A* pplication *C* onfiguration is a Python source file, generally with the *.tac* extension, which defines configuration to make an application runnable using `twistd` .

TAP

T wisted *A* pplication *P* ickle (no longer supported), or simply just a*T* wisted *AP* plication. A serialised application that was created with `mktap` (no longer supported) and runnable by `twistd` . See:doc:*Using the Utilities <basics>* .

Trial

twisted.trial , Twisted's unit-testing framework, based on the `unittest` standard library module. See also *Writing tests for Twisted code* .

Twisted Matrix Laboratories

The team behind Twisted. http://twistedmatrix.com/ .

Twisted Reality

In days of old, the Twisted Reality multiplayer text-based interactive-fiction system was the main focus of Twisted Matrix Labs; Twisted, the general networking framework, grew out of Reality's need for better network functionality. Twisted Reality has been superseded by the Imaginary project.

usage

The twisted.python.usage module, a replacement for the standard `getopt` module for parsing command-lines which is much easier to work with. See *Parsing command-lines* .

Words

Twisted Words is a multi-protocol chat server that uses the *Perspective Broker* protocol as its native communication style. See twisted.words .

Woven

W eb *O* bject *V* isualization *En* vironment. A templating system previously, but no longer, included with Twisted. Woven has largely been superseded by Divmod Nevow .

Debugging Python(Twisted) with Emacs

- Open up your project files. sometimes emacs can't find them if you don't have them open before-hand.

- Make sure you have a program called `pdb` somewhere in your PATH, with the following contents:

```
#!/bin/sh
exec python -m pdb $1 $2 $3 $4 $5 $6 $7 $8 $9
```

- Run `M-x pdb` in emacs. If you usually run your program as `python foo.py` , your command line should be `pdb foo.py` , for `twistd` and `trial` just add -b to the command line, e.g.: `twistd -b -y my.tac`
- While pdb waits for your input, go to a place in your code and hit `C-x SPC` to insert a break-point. pdb should say something happy. Do this in as many points as you wish.
- Go to your pdb buffer and hit `c` ; this runs as normal until a break-point is found.
- Once you get to a breakpoint, use `s` to step, `n` to run the current line without stepping through the functions it calls, `w` to print out the current stack, `u` and `d` to go up and down a level in the stack, `p foo` to print result of expression `foo` .
- Recommendations for effective debugging:
  - use `p self` a lot; just knowing the class where the current code is isn't enough most of the time.
  - use `w` to get your bearings, it'll re-display the current-line/arrow
  - after you use `w` , use `u` and `d` and lots more `p self` on the different stack-levels.
  - If you've got a big code-path that you need to grok, keep another buffer open and list the code-path there (e.g., I had a nasty-evil Deferred recursion, and this helped me tons)
- Introduction
  - *Executive summary*

    Connecting your software - and having fun too!
- Getting Started
  - *Writing a TCP server*

    Basic network servers with Twisted.
  - *Writing a TCP client*

    And basic clients.
  - *Test-driven development with Twisted*

    Code without tests is broken by definition; Twisted makes it easy to test your network code.
  - *Tutorial: Twisted From Scratch*
    1. *The Evolution of Finger: building a simple finger service*
    2. *The Evolution of Finger: adding features to the finger service*
    3. *The Evolution of Finger: cleaning up the finger code*
    4. *The Evolution of Finger: moving to a component based architecture*
    5. *The Evolution of Finger: pluggable backends*
    6. *The Evolution of Finger: a clean web frontend*
    7. *The Evolution of Finger: Twisted client support using Perspective Broker*
    8. *The Evolution of Finger: using a single factory for multiple protocols*
    9. *The Evolution of Finger: a Twisted finger client*
    10. *The Evolution of Finger: making a finger library*

11. *The Evolution of Finger: configuration and packaging of the finger service*

- *Setting up the TwistedQuotes application*
- *Designing a Twisted application*

- Networking and Other Event Sources
  - *Twisted Internet*

    A brief overview of the `twisted.internet` package.
  - *Reactor basics*

    The event loop at the core of your program.
  - *Using SSL in Twisted*

    Add some security to your network transport.
  - *UDP Networking*

    How to use Twisted's UDP implementation, including multicast and broadcast functionality.
  - *Using processes*

    Launching sub-processes, the correct way.
  - *Introduction to Deferreds*

    Like callback functions, only a lot better.
  - *Deferred reference*

    In-depth information on Deferreds.
  - *Generating deferreds*

    More about Deferreds.
  - *Scheduling*

    Timeouts, repeated events, and more: when you want things to happen later.
  - *Using threads*

    Running code in threads, and interacting with Twisted in a thread-safe manner.
  - *Producers and Consumers: Efficient High-Volume Streaming*

    How to pause when buffers fill up.
  - *Choosing a reactor and GUI toolkit integration*

    GTK+, Windows, epoll() and more: use your GUI of choice, or a faster event loop.
- High-Level Infrastructure
  - *Getting Connected with Endpoints*

    Create configurable applications that support multiple transports (e.g. TCP and SSL).
  - *Interfaces and Adapters (Component Architecture)*

    When inheritance isn't enough.
  - *Cred: Pluggable Authentication*

    Implementing authentication and authorization that is configurable, pluggable and re-usable.

  - *Twisted's plugin architecture*

    A generic plugin system for extendable programs.

- Deploying Twisted Applications
  - *Helper programs and scripts (twistd, ..)*

    `twistd` lets you daemonize and run your application.
  - *Using the Twisted Application Framework*

    Writing code that `twistd` can run.
  - *Writing Twisted Application Plugins for twistd*

    More powerful `twistd` deployment method.
  - *Deploying Twisted with systemd*

    Use `systemd` to launch and monitor Twisted applications.
- Utilities
  - *Emitting and Observing Logs*

    Keep a record of what your application is up to, and inspect that record to discover interesting information. (You may also be interested in the *legacy logging system* if you are maintaining code written to work with older versions of Twisted.)
  - *Symbolic constants*

    enum-like constants. (Deprecated, spun out into Constantly)
  - *Twisted RDBMS support with adbapi*

    Using SQL with your relational database via DB-API adapters.
  - *Parsing command-line arguments*

    The command-line argument parsing used by `twistd`.
  - *Using Dirdbm: Directory-based Storage*

    A simplistic way to store data on your filesystem.
  - *Tips for writing tests for Twisted code using Trial*

    More information on writing tests.
  - *Extremely Low-Level Socket Operations*

    Using wrappers for sendmsg(2) and recvmsg(2).
- Asynchronous Messaging Protocol (AMP)
  - *Asynchronous Messaging Protocol Overview*

    A two-way asynchronous message passing protocol, for when HTTP isn't good enough.
- Perspective Broker
  - *Twisted Spread*

    A remote method invocation (RMI) protocol: call methods on remote objects.
  - *Introduction to Perspective Broker*
  - *Using Perspective Broker*
  - *Managing Clients of Perspectives*

Examples

Simple Echo server and client

- `simpleclient.py` - simple TCP client
- `simpleserv.py` - simple TCP echo server

Chat

- `chatserver.py` - shows how to communicate between clients

Echo server & client variants

- `echoserv.py` - variant on a simple TCP echo server
- `echoclient.py` - variant on a simple TCP client
- `echoserv_udp.py` - simplest possible UDP server
- `echoclient_udp.py` - simple UDP client
- `echoserv_ssl.py` - simple SSL server
- `echoclient_ssl.py` - simple SSL client

AMP server & client variants

- `ampserver.py` - do math using AMP
- `ampclient.py` - do math using AMP

Perspective Broker

- `pbsimple.py` - simplest possible PB server
- `pbsimpleclient.py` - simplest possible PB client
- `pbbenchclient.py` - benchmarking client

- `pbbenchserver.py` - benchmarking server
- `pbecho.py` - echo server that uses login
- `pbechoclient.py` - echo client using login
- `pb_exceptions.py` - example of exceptions over PB
- `pbgtk2.py` - example of using GTK2 with PB
- `pbinterop.py` - shows off various types supported by PB
- `bananabench.py` - benchmark for banana

Cred

- `cred.py` - Authenticate a user with an in-memory username/password database
- `dbcred.py` - Using a database backend to authenticate a user

GUI

- `wxdemo.py` - demo of wxPython integration with Twisted
- `pbgtk2.py` - example of using GTK2 with PB
- `pyuidemo.py` - PyUI

FTP examples

- `ftpclient.py` - example of using the FTP client
- `ftpserver.py` - create an FTP server which serves files for anonymous users from the working directory and serves files for authenticated users from `/home`.

Logging

- `twistd-logging.tac` - logging example using ILogObserver
- `testlogging.py` - use twisted.python.log to log errors to standard out
- `rotatinglog.py` - example of log file rotation

POSIX Specific Tricks

- `sendfd.py`, `recvfd.py` - send and receive file descriptors over UNIX domain sockets

Miscellaneous

- `shaper.py` - example of rate-limiting your web server
- `stdiodemo.py` - example using stdio, Deferreds, LineReceiver and twisted.web.client.
- `ptyserv.py` - serve shells in pseudo-terminals over TCP
- `courier.py` - example of interfacing to Courier's mail filter interface

- `longex.py` - example of doing arbitrarily long calculations nicely in Twisted
- `longex2.py` - using generators to do long calculations
- `stdin.py` - reading a line at a time from standard input without blocking the reactor
- `streaming.py` - example of a push producer/consumer system
- `filewatch.py` - write the content of a file to standard out one line at a time
- `shoutcast.py` - example Shoutcast client
- `wxacceptance.py` - acceptance tests for wxreactor
- `postfix.py` - test application for PostfixTCPMapServer
- `udpbroadcast.py` - broadcasting using UDP
- `tls_alpn_npn_client.py` - example of TLS next-protocol negotiation on the client side using NPN and ALPN.
- `tls_alpn_npn_server.py` - example of TLS next-protocol negotiation on the server side using NPN and ALPN.

Specifications

Banana Protocol Specifications

Introduction

Banana is an efficient, extendable protocol for sending and receiving s-expressions. A s-expression in this context is a list composed of bytes, integers, large integers, floats and/or s-expressions. Unicode is not supported (but can be encoded to and decoded from bytes on the way into and out of Banana). Unsupported types must be converted into a supported type before sending them with Banana.

Banana Encodings

The banana protocol is a stream of data composed of elements. Each element has the following general structure - first, the length of element encoded in base-128, least significant bit first. For example length 4674 will be sent as `0x42 0x24` . For certain element types the length will be omitted (e.g. float) or have a different meaning (it is the actual value of integer elements).

Following the length is a delimiter byte, which tells us what kind of element this is. Depending on the element type, there will then follow the number of bytes specified in the length. The byte's high-bit will always be set, so that we can differentiate between it and the length (since the length bytes use 128-base, their high bit will never be set).

Element Types

Given a series of bytes that gave us length N, these are the different delimiter bytes:

List – 0x80

> The following bytes are a list of N elements. Lists may be nested, and a child list counts as only one element to its parent (regardless of how many elements the child list contains).

Integer – 0x81

The value of this element is the positive integer N. Following bytes are not part of this element. Integers can have values of 0 <= N <= 2147483647.

String – 0x82

The following N bytes are a string element.

Negative Integer – 0x83

The value of this element is the integer N * -1, i.e. -N. Following bytes are not part of this element. Negative integers can have values of 0 >= -N >= -2147483648.

Float - 0x84

The next 8 bytes are the float encoded in IEEE 754 floating-point "double format" bit layout. No length bytes should have been defined.

Large Integer – 0x85

The value of this element is the positive large integer N. Following bytes are not part of this element. Large integers have no size limitation.

Large Negative Integer – 0x86

The value of this element is the negative large integer -N. Following bytes are not part of this element. Large integers have no size limitation.

Large integers are intended for arbitrary length integers. Regular integers types (positive and negative) are limited to 32-bit values.

Examples

Here are some examples of elements and their encodings - the type bytes are marked in bold:

```
1
    0x01 **0x81**
-1
    0x01 **0x83**
1.5
    **0x84** 0x3f 0xf8 0x00 0x00 0x00 0x00 0x00 0x00
"hello"
    0x05 **0x82** 0x68 0x65 0x6c 0x6c 0x6f
[]
    0x00 **0x80**
[1, 23]
    0x02 **0x80** 0x01 **0x81** 0x17 **0x81**
123456789123456789
    0x15 0x3e 0x41 0x66 0x3a 0x69 0x26 0x5b 0x01 **0x85**
[1, ["hello"]]
    0x02 **0x80** 0x01 **0x81** 0x01 **0x80** 0x05 **0x82** 0x68 0x65
    0x6c 0x6c 0x6f
```

Profiles

The Banana protocol is extendable. Therefore, it supports the concept of profiles. Profiles allow developers to extend the banana protocol, adding new element types, while still keeping backwards compatibility with implementations that don't support the extensions. The profile used in each session is determined at the handshake stage (see below.)

A profile is specified by a unique string. This specification defines two profiles - `"none"` and `"pb"` . The `"none"` profile is the standard profile that should be supported by all Banana implementations. Additional profiles may be added in the future.

Extensions defined by a profile may only be used if that profile has been selected by client and server.

The `"none"` Profile

The `"none"` profile is identical to the delimiter types listed above. It is highly recommended that all Banana clients and servers support the `"none"` profile.

The `"pb"` Profile

The `"pb"` profile is intended for use with the Perspective Broker protocol, that runs on top of Banana. Basically, it converts commonly used PB strings into shorter versions, thus minimizing bandwidth usage. It starts with a single byte, which tells us to which string element to convert it, and ends with the delimiter byte, `0x87` , which should not be prefixed by a length.

0x01 'None'

0x02 'class'

0x03 'dereference'

0x04 'reference'

0x05 'dictionary'

0x06 'function'

0x07 'instance'

0x08 'list'

0x09 'module'

0x0a 'persistent'

0x0b 'tuple'

0x0c 'unpersistable'

0x0d 'copy'

0x0e 'cache'

0x0f 'cached'

0x10 'remote'

0x11 'local'

0x12 'lcache'

0x13 'version'

0x14 'login'

0x15 ‘password’

0x16 ‘challenge’

0x17 ‘logged_in’

0x18 ‘not_logged_in’

0x19 ‘cachemessage’

0x1a ‘message’

0x1b ‘answer’

0x1c ‘error’

0x1d ‘decref’

0x1e ‘decache’

0x1f ‘uncache’

Protocol Handshake and Behaviour

The initiating side of the connection will be referred to as “client” , and the other side as “server” .

Upon connection, the server will send the client a list of string elements, signifying the profiles it supports. It is recommended that `"none"` be included in this list. The client then sends the server a string from this list, telling the server which profile it wants to use. At this point the whole session will use this profile.

Once a profile has been established, the two sides may start exchanging elements. There is no limitation on order or dependencies of messages. Any such limitation (e.g. “server can only send an element to client in response to a request from client”) is application specific.

Upon receiving illegal messages, failed handshakes, etc., a Banana client or server should close its connection.

- *Banana Protocol Specifications*

Development of Twisted

Naming Conventions

While this may sound like a small detail, clear method naming is important to provide an API that developers familiar with event-based programming can pick up quickly.

Since the idea of a method call maps very neatly onto that of a received event, all event handlers are simply methods named after past-tense verbs. All class names are descriptive nouns, designed to mirror the is-a relationship of the abstractions they implement. All requests for notification or transmission are present-tense imperative verbs.

Here are some examples of this naming scheme:

- An event notification of data received from peer:`dataReceived(data)`
- A request to send data: `write(data)`
- A class that implements a protocol: `Protocol`

The naming is platform neutral. This means that the names are equally appropriate in a wide variety of environments, as long as they can publish the required events.

It is self-consistent. Things that deal with TCP use the acronym TCP, and it is always capitalized. Dropping, losing, terminating, and closing the connection are all referred to as "losing" the connection. This symmetrical naming allows developers to easily locate other API calls if they have learned a few related to what they want to do.

It is semantically clear. The semantics of dataReceived are simple: there are some bytes available for processing. This remains true even if the lower-level machinery to get the data is highly complex.

Philosophy

Abstraction Levels

When implementing interfaces to the operating system or the network, provide two interfaces:

- One that doesn't hide platform specific or library specific functionality. For example, you can use file descriptors on Unix, and Win32 events on Windows.
- One that provides a high level interface hiding platform specific details. E.g. process running uses same API on Unix and Windows, although the implementation is very different.

Restated in a more general way:

- Provide all low level functionality for your specific domain, without limiting the policies and decisions the user can make.
- Provide a high level abstraction on top of the low level implementation (or implementations) which implements the common use cases and functionality that is used in most cases.

Learning Curves

Require the minimal amount of work and learning on part of the user to get started. If this means they have less functionality, that's OK, when they need it they can learn a bit more. This will also lead to a cleaner, easier to test design.

For example - using twistd is a great way to deploy applications. But to get started you don't need to know about it. Later on you can start using twistd, but its usage is optional.

Security

We need to do a full audit of Twisted, module by module. This document list the sort of things you want to look for when doing this, or when writing your own code.

Bad input

Any place we receive untrusted data, we need to be careful. In some cases we are not careful enough. For example, in HTTP there are many places where strings need to be converted to ints, so we use `int()` . The problem is that this well accept negative numbers as well, whereas the protocol should only be accepting positive numbers.

Resource Exhaustion and DoS

Make sure we never allow users to create arbitrarily large strings or files. Some of the protocols still have issues like this. Place a limit which allows reasonable use but will cut off huge requests, and allow changing of this limit.

Another operation to look out for are exceptions. They can fill up logs and take a lot of CPU time to render in web pages.

Twisted Development Policy

Twisted Coding Standard

Naming

Try to choose names which are both easy to remember and meaningful. Some silliness is OK at the module naming level (see twisted.spread ...) but when choosing class names, be as precise as possible.

Try to avoid terms that may have existing definitions or uses. This rule is often broken, since it is incredibly difficult, as most normal words have already been taken by some other software. As an example, using the term "reactor" elsewhere in Twisted for something that is not an implementor of `IReactor` adds additional meaning to the word and will cause confusion.

More importantly, try to avoid meaningless words. In particular, words like "handler", "processor", "engine", "manager", and "component" don't really indicate what something does, only that it does *something*.

Use American spelling in both names and docstrings. For compound technical terms such as 'filesystem', use a non-hyphenated spelling in both docstrings and code in order to avoid unnecessary capitalization.

Testing

Overview

Twisted development should always be test-driven . The complete test suite in the head of the Git trunk is required to be passing on supported platforms at all times. Regressions in the test suite are addressed by reverting whatever revisions introduced them.

Test Suite

Note: The *test standard* contains more in-depth information on this topic. What follows is intended to be a synopsis of the most important points.

The Twisted test suite is spread across many subpackages of the `twisted` package. Many older tests are in `twisted.test` . Others can be found at places such as `twisted.web.test` (for `twisted.web` tests) or `twisted.internet.test` (for `twisted.internet` tests). The latter arrangement, `twisted.somepackage.test`, is preferred for new tests except when a test module already exists in `twisted.test` .

Parts of the Twisted test suite may serve as good examples of how to write tests for Twisted or for Twisted-based libraries (newer parts of the test suite are generally better examples than older parts - check when the code you are looking at was written before you use it as an example of what you should write). The names of test modules must begin with `test_` so that they are automatically discoverable by test runners such as Trial. Twisted's unit tests are written using twisted.trial, an xUnit library which has been extensively customized for use in testing Twisted and Twisted-based libraries.

Implementation (i.e., non-test) source files should begin with a `test-case-name` tag which gives the name of any test modules or packages which exercise them. This lets tools discover a subset of the entire test suite which they can run first to find tests which might be broken by a particular change.

All unit test methods should have docstrings specifying at a high level the intent of the test. That is, a description that users of the method would understand.

If you modify, or write a new, HOWTO, please read the *documentation writing standard*.

Copyright Header

Whenever a new file is added to the repository, add the following license header at the top of the file:

```
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.
```

When you update existing files, if there is no copyright header, add one.

Whitespace

Indentation is 4 spaces per indent. Tabs are not allowed. It is preferred that every block appears on a new line, so that control structure indentation is always visible.

Lines are flowed at 79 columns. They must not have trailing whitespace. Long lines must be wrapped using implied line continuation inside parentheses; backslashes aren't allowed. To handle long import lines, please wrap them inside parentheses:

```
from very.long.package import (foo, bar, baz,
                               qux, quux, quuux)
```

Top-level classes and functions must be separated with 3 blank lines, and class-level functions with 2 blank lines. The control-L (i.e. `^L`) form feed character must not be used.

Modules

Modules must be named in all lower-case, preferably short, single words. If a module name contains multiple words, they may be separated by underscores or not separated at all.

Modules must have a copyright message, a docstring, and a reference to a test module that contains the bulk of its tests. New modules must have the `absolute_import`, `division`, and optionally the `print_function` imports from the `__future__` module.

Use this template:

`new_module_template.py`

```
# -*- test-case-name: <test module> -*-
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

"""
Docstring goes here.
"""

from __future__ import absolute_import, division, print_function


__all__ = []
```

In most cases, modules should contain more than one class, function, or method; if a module contains only one object, consider refactoring to include more related functionality in that module.

Depending on the situation, it is acceptable to have imports that look like this:

```
from twisted.internet.defer import Deferred
```

or like this:

```
from twisted.internet import defer
```

That is, modules should import *modules* or *classes and functions*, but not *packages*.

Wildcard import syntax may not be used by code in Twisted. These imports lead to code which is difficult to read and maintain by introducing complexity which strains human readers and automated tools alike. If you find yourself with many imports to make from a single module and wish to save typing, consider importing the module itself, rather than its attributes.

Relative imports (or *sibling imports*) may not be used by code in Twisted. Relative imports allow certain circularities to be introduced which can ultimately lead to unimportable modules or duplicate instances of a single module. Relative imports also make the task of refactoring more difficult.

In case of local names conflicts due to import, use the `as` syntax, for example:

```
from twisted.trial import util as trial_util
```

The encoding must always be ASCII, so no coding cookie is necessary.

Packages

Package names follow the same conventions as module names. All modules must be encapsulated in some package. Nested packages may be used to further organize related modules.

`__init__.py` must never contain anything other than a docstring and (optionally) an `__all__` attribute. Packages are not modules and should be treated differently. This rule may be broken to preserve backwards compatibility if a module is made into a nested package as part of a refactoring.

If you wish to promote code from a module to a package, for example, to break a large module out into several smaller files, the accepted way to do this is to promote from within the module. For example,

```
# parent/
# --- __init__.py ---
import child

# --- child.py ---
import parent
class Foo:
    pass
parent.Foo = Foo
```

Packages must not depend circularly upon each other. To simplify maintaining this state, packages must also not import each other circularly. While this applies to all packages within Twisted, one `twisted.python` deserves particular attention, as it may not depend on any other Twisted package.

Strings

All strings in Twisted which are not interfacing directly with Python (e.g. `sys.path` contents, module names, and anything which returns `str` on both Python 2 and 3) should be marked explicitly as "bytestrings" or "text/Unicode strings". This is done by using the `b` (for bytestrings) or `u` (for Unicode strings) prefixes when using string literals.

String literals not marked with this are "native/bare strings", and have a different meaning on Python 2 (where a bare string is a bytestring) and Python 3 (where a bare string is a Unicode string).

```
u"I am text, also known as a Unicode string!"
b"I am a bytestring!"
"I am a native bare string, and therefore may be either!"
```

Bytestrings and text must not be implicitly concatenated, as this causes an invisible ASCII encode/decode on Python 2, and causes an exception on Python 3.

Use + to combine bytestrings, not string formatting (either "percent formatting" or `.format()`). String formatting is not available on Python 3.3 and 3.4.

```
HTTPVersion = b"1.1"
transport.write(b"HTTP/" + HTTPVersion)
```

Utilities are available in twisted.python.compat to paper over some use cases where other Python code (especially the standard library) expects a "native string", or provides a native string where a bytestring is actually required (namely twisted.python.compat.nativeString and twisted.python.compat.networkString)

String Formatting Operations

String formatting operations like `formatString % values` should only be used on text strings, not byte strings, as they do not work on Python 3.3 and 3.4.

When using "percent formatting", you should always use a tuple if you're using non-mapping `values`. This is to avoid unexpected behavior when you think you're passing in a single value, but the value is unexpectedly a tuple, e.g.:

```
def foo(x):
    return "Hi %s\n" % x
```

The example shows you can pass in `foo("foo")` or `foo(3)` fine, but if you pass in `foo((1,2))`, it raises a `TypeError`. You should use this instead:

```
def foo(x):
    return "Hi %s\n" % (x,)
```

Docstrings

Docstrings should always be used to describe the purpose of methods, functions, classes, and modules. Moreover, all methods, functions, classes, and modules must have docstrings. In addition to documenting the purpose of the object, the docstring must document all of parameters or attributes of the object.

When documenting a class with one or more attributes which are initialized directly from the value of a `__init__` argument by the same name (or differing only in that the attribute is private), it is sufficient to document the `__init__` parameter (in the `__init__` docstring). For example:

```
class Ninja(object):
    """
    @ivar speed: See L{__init__}
    @ivar _stealth: See C{stealth} parameter of L{__init__}
    """
    def __init__(self, speed, stealth):
        """
        @param speed: The maximum rate at which this ninja can travel (m/s)
```

```
        @type speed: L{int} or L{float}

        @param stealth: This ninja's ability to avoid being noticed in its
            activities, as a percentage modifier.
        @type: L{int}
        """
        self.speed = speed
        self._stealth = stealth
```

It is not necessary to have a second copy of the documentation for the attribute in the class docstring, only a reference to the method (typically `__init__` which does document the attribute's meaning). Of course, if there is any interesting additional behavior about the attribute that does not apply to the `__init__` argument, that behavior should be documented in the class docstring.

Docstrings are *never* to be used to provide semantic information about an object; this rule may be violated if the code in question is to be used in a system where this is a requirement (such as Zope).

Docstrings should be indented to the level of the code they are documenting.

Docstrings must be triple-quoted, with opening and the closing of the docstrings being on a line by themselves. For example:

```
class Ninja(object):
    """
    A L{Ninja} is a warrior specializing in various unorthodox arts of war.
    """
    def attack(self, someone):
        """
        Attack C{someone} with this L{Ninja}'s shuriken.
        """
```

Docstrings are written in epytext format; more documentation is available in the Epytext Markup Language documentation.

When you are referring to a type, you should use *L{}*, whether it's in the stdlib , in Twisted or somewhere else.

NoneType is an exception and we are referring it just as *L{None}*.

Pydoctor, the software we use to generate the documentation, links to the Python standard library if you use `L{}` with standard Python types (e.g. `L{str}`).

For the API doc *C{something}* means "I made up a new word, and I want it to be monospaced, like it's an identifier in code and not an English noun"

L{something} means "I am referring to the previously-defined concept/package/module/class/function/method/attribute identified as *something*"

Additionally, to accommodate emacs users, single quotes of the type of the docstring's triple-quote should be escaped. This will prevent font-lock from accidentally fontifying large portions of the file as a string.

For example,

```
def foo2bar(f):
    """
    Convert L{foo}s to L{bar}s.

    A function that should be used when you have a C{foo} but you want a
    C{bar}; note that this is a non-destructive operation.  If this method
    can't convert the C{foo} to a C{bar} it will raise a L{FooException}.

    @param f: C{foo}
```

```
    @type f: L{str}

    For example::

        import wombat
        def sample(something):
            f = something.getFoo()
            f.doFooThing()
            b = wombat.foo2bar(f)
            b.doBarThing()
            return b

    """
    # Optionally, actual code can go here.
```

Comments

Start by reading the PEP8 Comments section. Ignore *Documentation Strings* section from PEP8 as Twisted uses a different docstring standard.

FIXME/TODO comments must have an associated ticket and contain a reference to it in the form of a full URL to the ticket. A brief amount of text should provide info about why the FIXME was added. It does not have to be the full ticket description, just enough to help readers decide if their next step should be reading the ticket or continue reading the code.

```
# FIXME: https://twistedmatrix.com/trac/ticket/1235
# Threads that have died before calling stop() are not joined.
for thread in threads:
    thread.join()
```

Versioning

The API documentation should be marked up with version information. When a new API is added the class should be marked with the epytext `@since:` field including the version number when the change was introduced. This version number should be in the form `x.y` (for example, `@since:  15.1`).

Scripts

For each "script" , that is, a program you expect a Twisted user to run from the command-line, the following things must be done:

1. Write a module in twisted.scripts which contains a callable global named `run`. This will be called by the command line part with no arguments (it will usually read `sys.argv`). Feel free to write more functions or classes in this module, if you feel they are useful to others.
2. Create a file which contains a shebang line for Python. This file should be placed in the `bin/` directory; for example, `bin/twistd`.

   ```
   #!/usr/bin/env python
   ```

To make sure that the script is portable across different UNIX like operating systems we use the `/usr/bin/env` command. The env command allows you to run a program in a modified environment. That way you don't have to search for a program via the `PATH` environment variable. This makes the script more portable but note that it is not

a foolproof method. Always make sure that `/usr/bin/env` exists or use a softlink/symbolic link to point it to the correct path. Python's distutils will rewrite the shebang line upon installation so this policy only covers the source files in version control.

1. For core scripts, add an entry to `_CONSOLE_SCRIPTS` in `src/twisted/python/_setup.py`:

```
_CONSOLE_SCRIPTS = [
    ...
    "twistd = twisted.scripts.twistd:run",
    "yourmodule" = "twisted.scripts.yourmodule:run",
]
```

2. And end with:

```
from twisted.scripts.yourmodule import run
run()
```

3. Write a manpage and add it to the `man` folder of a subproject's `doc` folder. On Debian systems you can find a skeleton example of a manpage in `/usr/share/doc/man-db/examples/manpage.example`.

This will ensure that your program will have a proper `console_scripts` entry point, which `setup.py` will use to generate a console script which will work correctly for users of Git, Windows releases and Debian packages.

Examples

For example scripts you expect a Twisted user to run from the command-line, add this Python shebang line at the top of the file:

```
#!/usr/bin/env python
```

Standard Library Extension Modules

When using the extension version of a module for which there is also a Python version, place the import statement inside a try/except block, and import the Python version if the import fails. This allows code to work on platforms where the extension version is not available. For example:

```
try:
    import cPickle as pickle
except ImportError:
    import pickle
```

Use the "as" syntax of the import statement as well, to set the name of the extension module to the name of the Python module.

Some modules don't exist across all supported Python versions. For example, Python 2.3's `sets` module was deprecated in Python 2.6 in favor of the `set` and `frozenset` builtins. twisted.python.compat would be the place to add `set` and `frozenset` implementations that work across Python versions.

Classes

Classes are to be named in mixed case, with the first letter capitalized; each word separated by having its first letter capitalized. Acronyms should be capitalized in their entirety. Class names should not be prefixed with the name of the module they are in. Examples of classes meeting this criteria:

- `twisted.spread.pb.ViewPoint`
- `twisted.parser.patterns.Pattern`

Examples of classes **not** meeting this criteria:

- `event.EventHandler`
- `main.MainGadget`

An effort should be made to prevent class names from clashing with each other between modules, to reduce the need for qualification when importing. For example, a Service subclass for Forums might be named `twisted.forum.service.ForumService`, and a Service subclass for Words might be `twisted.words.service.WordsService`. Since neither of these modules are volatile *(see above)* the classes may be imported directly into the user's namespace and not cause confusion.

New-style Classes

Classes and instances in Python come in two flavors: old-style or classic, and new-style. Up to Python 2.1, old-style classes were the only flavour available to the user, new-style classes were introduced in Python 2.2 to unify classes and types. All classes added to Twisted must be written as new-style classes. If `x` is an instance of a new-style class, then `type(x)` is the same as `x.__class__`.

Methods

Methods should be in mixed case, with the first letter lower case, each word separated by having its first letter capitalized. For example, `someMethodName`, `method`.

Sometimes, a class will dispatch to a specialized sort of method using its name; for example, `twisted.reflect.Accessor`. In those cases, the type of method should be a prefix in all lower-case with a trailing underscore, so method names will have an underscore in them. For example, `get_someAttribute`. Underscores in method names in twisted code are therefore expected to have some semantic associated with them.

Some methods, in particular `addCallback` and its cousins return self to allow for chaining calls. In this case, wrap the chain in parenthesis, and start each chained call on a separate line, for example:

```
return (foo()
        .addCallback(bar)
        .addCallback(thud)
        .addCallback(wozers))
```

Using the Global Reactor

Even though it may be convenient, module-level imports of the global Twisted reactor (`from twisted.internet import reactor`) should be avoided. Importing the reactor at the module level means that reactor selection occurs on initial import, and not at the request of the code that originally imported the module. Applications may wish to import their own reactor, or otherwise use a reactor different than Twisted's default (for example, using the experimental cfreactor on OS X); importing at the module level means they would have to monkeypatch in the different reactor, or use similar hacks. This is especially apparent in Twisted's own test suite; many tests wish to provide their own reactor which controls the passage of time and simulates timeouts.

Below is an example of the pattern for accepting the user's choice of reactor – importing the global one if none is specified – taken (and trimmed for brevity) from existing Twisted source code.

```
class Session(object):
    """
    A user's session with a system.

    @ivar _reactor: An object providing L{IReactorTime} to use for scheduling
        expiration.
    """
    def __init__(self, site, uid, reactor=None):
        """
        Initialize a session with a unique ID for that session.
        """
        if reactor is None:
            from twisted.internet import reactor
        self._reactor = reactor

        # ... other code ...
```

The reactor attribute should be private by default, but if it is useful to the users of the code, there is no reason why it can not be public.

Callback Arguments

There are several methods whose purpose is to help the user set up callback functions, for example Deferred.addCallback or the reactor's callLater method. To make access to the callback as transparent as possible, most of these methods use `**kwargs` to capture arbitrary arguments that are destined for the user's callback. This allows the call to the setup function to look very much like the eventual call to the target callback function.

In these methods, take care to not have other argument names that will "steal" the user's callback's arguments. When sensible, prefix these "internal" argument names with an underscore. For example, RemoteReference.callRemote is meant to be called like this:

```
myref.callRemote("addUser", "bob", "555-1212")

# on the remote end, the following method is invoked:
def addUser(name, phone):
    ...
```

where "addUser" is the remote method name. The user might also choose to call it with named parameters like this:

```
myref.callRemote("addUser", name="bob", phone="555-1212")
```

In this case, `callRemote` (and any code that uses the `**kwargs` syntax) must be careful to not use "name", "phone", or any other name that might overlap with a user-provided named parameter. Therefore, `callRemote` is implemented with the following signature:

```
class SomeClass(object):
    def callRemote(self, _name, *args, **kw):
        ...
```

Do whatever you can to reduce user confusion. It may also be appropriate to `assert` that the kwargs dictionary does not contain parameters with names that will eventually cause problems.

Special Methods

The augmented assignment protocol, defined by `__iadd__` and other similarly named methods, can be used to allow objects to be modified in place or to rebind names if an object is immutable – both through use of the same operator. This can lead to confusing code, which in turn leads to buggy code. For this reason, methods of the augmented assignment protocol should not be used in Twisted.

Functions

Functions should be named similarly to methods.

Functions or methods which are responding to events to complete a callback or errback should be named `_cbMethodName` or `_ebMethodName`, in order to distinguish them from normal methods.

Attributes

Attributes should be named similarly to functions and methods. Attributes should be named descriptively; attribute names like `mode`, `type`, and `buf` are generally discouraged. Instead, use `displayMode`, `playerType`, or `inputBuffer`.

Do not use Python's "private" attribute syntax; prefix non-public attributes with a single leading underscore. Since several classes have the same name in Twisted, and they are distinguished by which package they come from, Python's double-underscore name mangling will not work reliably in some cases. Also, name-mangled private variables are more difficult to address when unit testing or persisting a class.

An attribute (or function, method or class) should be considered private when one or more of the following conditions are true:

- The attribute represents intermediate state which is not always kept up-to-date.
- Referring to the contents of the attribute or otherwise maintaining a reference to it may cause resources to leak.
- Assigning to the attribute will break internal assumptions.
- The attribute is part of a known-to-be-sub-optimal interface and will certainly be removed in a future release.

Python 3

Twisted is being ported to Python 3, targeting Python 3.3+. Please see *Porting to Python 3* for details.

All new modules must be Python 2.7 & 3.3+ compatible, and all new code to ported modules must be Python 2.7 & 3.3+ compatible. New code in non-ported modules must be written in a 2.7 & 3.3+ compatible way (explicit bytes/unicode strings, new exception raising format, etc) as to prevent extra work when that module is eventually ported. Code targeting Python 3 specific features must gracefully fall-back on Python 2 as much as is reasonably possible (for example, Python 2 support for 'async/await' is not reasonably possible and would not be required, but code that uses a Python 3-specific module such as ipaddress should be able to use a backport to 2.7 if available).

Database

Database tables will be named with plural nouns.

Database columns will be named with underscores between words, all lower case, since most databases do not distinguish between case.

Any attribute, method argument, or method name that corresponds *directly* to a column in the database will be named exactly the same as that column, regardless of other coding conventions surrounding that circumstance.

All SQL keywords should be in upper case.

C Code

C code must be optional, and work across multiple platforms (MSVC++9/10/14 for Pythons 2.7, 3.3/3.4, and 3.5 on Windows, as well as recent GCCs and Clangs for Linux, OS X, and FreeBSD).

C code should be kept in external bindings packages which Twisted depends on. If creating new C extension modules, using cffi is highly encouraged, as it will perform well on PyPy and CPython, and be easier to use on Python 2 and 3. Consider optimizing for PyPy instead of creating bespoke C code.

Commit Messages

The commit messages are being distributed in a myriad of ways. Because of that, you need to observe a few simple rules when writing a commit message.

The first line of the message is being used as both the subject of the commit email and the announcement on #twisted. Therefore, it should be short (aim for < 80 characters) and descriptive – and must be able to stand alone (it is best if it is a complete sentence). The rest of the e-mail should be separated with *hard line breaks* into short lines (< 70 characters). This is free-format, so you can do whatever you like here.

Commit messages should be about *what*, not *how*: we can get how from Git diff. Explain reasons for commits, and what they affect.

Each commit should be a single logical change, which is internally consistent. If you can't summarize your changes in one short line, this is probably a sign that they should be broken into multiple checkins.

Source Control

Twisted currently uses Git for source control. All development must occur using branches; when a task is considered complete another Twisted developer may review it and if no problems are found, it may be merged into trunk. The Twisted wiki has a start.

If you wish to ignore certain files, create a `.gitignore` file, or edit it if it exists. For example:

```
dropin.cache
*.pyc
*.pyo
*.o
*.lo
*.la #*#
.*.rej
*.rej
.*~
```

Fallback

In case of conventions not enforced in this document, the reference documents to use in fallback is PEP 8 for Python code and PEP 7 for C code. For example, the paragraph **Whitespace in Expressions and Statements** in PEP 8 describes what should be done in Twisted code.

Recommendations

These things aren't necessarily standardizeable (in that code can't be easily checked for compliance) but are a good idea to keep in mind while working on Twisted.

If you're going to work on a fragment of the Twisted codebase, please consider finding a way that you would *use* such a fragment in daily life. Using a Twisted Web server on your website encourages you to actively maintain and improve your code, as the little everyday issues with using it become apparent. Twisted is a **big** codebase! If you're refactoring something, please make sure to recursively grep for the names of functions you're changing. You may be surprised to learn where something is called. Especially if you are moving or renaming a function, class, method, or module, make sure that it won't instantly break other code.

Twisted Writing Standard

The Twisted writing standard describes the documentation writing styles we prefer in our narrative documentation.

This standard applies particularly to howtos and other descriptive documentation. For writing API documentation, please refer to *Docstrings section in our coding standard*.

This document is meant to help Twisted documentation authors produce documentation that does not have the following problems:

- misleads users about what is good Twisted style;
- misleads users into thinking that an advanced howto is an introduction to writing their first Twisted server; and
- misleads users about whether they fit the document's target audience: for example, that they are able to use enterprise without knowing how to write SQL queries.

General style

Documents should aim to be clear and concise, allowing the API documentation and the example code to tell as much of the story as they can. Demonstrations and where necessary supported arguments should always preferred to simple statements ("here is how you would simplify this code with Deferreds" rather than "Deferreds make code simpler").

Documents should be clearly delineated into sections and subsections. Each of these sections, like the overall document, should have a single clear purpose. This is most easily tested by trying to have meaningful headings: a section which is headed by "More details" or "Advanced stuff" is not purposeful enough. There should be fairly obvious ways to split a document. The two most common are task based sectioning and sectioning which follows module and class separations.

Documentation must use American English spelling, and where possible avoid any local variants of either vocabulary or grammar. Grammatically complex sentences should ideally be avoided: these make reading unnecessarily difficult, particularly for non-native speakers.

When referring to a hypothetical person, (such as "a user of a website written with twisted.web"), gender neutral pronouns (they/their/them) should be used.

For reStructuredText documents which are handled by the Sphinx documentation generator make lines short, and break lines at natural places, such as after commas and semicolons, rather than after the 79th column. We call this *semantic newlines*. This rule **does not** apply to docstrings.

```
Sometimes when editing a narrative documentation file, I wrap the lines semantically.
Instead of inserting a newline at 79 columns (or whatever),
or making paragraphs one long line,
I put in newlines at a point that seems logical to me.
Modern code-oriented text editors are very good at wrapping and arranging long lines.
```

Evangelism and usage documents

The Twisted documentation should maintain a reasonable distinction between "evangelism" documentation, which compares the Twisted design or Twisted best practice with other approaches and argues for the Twisted approach, and "usage" documentation, which describes the Twisted approach in detail without comparison to other possible approaches.

While both kinds of documentation are useful, they have different audiences. The first kind of document, evangelical documents, is useful to a reader who is researching and comparing approaches and seeking to understand the Twisted approach or Twisted functionality in order to decide whether it is useful to them. The second kind of document, usage documents, are useful to a reader who has decided to use Twisted and simply wants further information about available functions and architectures they can use to accomplish their goal.

Since they have distinct audiences, evangelism and detailed usage documentation belongs in separate files. There should be links between them in 'Further reading' or similar sections.

Descriptions of features

Descriptions of any feature added since release 2.0 of Twisted core must have a note describing which release of which Twisted project they were added in at the first mention in each document. If they are not yet released, give them the number of the next minor release.

For example, a substantial change might have a version number added in the introduction:

> This document describes the Application infrastructure for deploying Twisted applications *(added in Twisted 1.3)* .

The version does not need to be mentioned elsewhere in the document except for specific features which were added in subsequent releases, which might should be mentioned separately.

> The simplest way to create a `.tac` file, SuperTac *(added in Twisted Core 99.7)* ...

In the case where the usage of a feature has substantially changed, the number should be that of the release in which the current usage became available. For example:

> This document describes the Application infrastructure for deploying Twisted applications *(updated[/substantially updated] in Twisted 2.7)* .

Linking

The first occurrence of the name of any module, class or function should always link to the API documents. Subsequent mentions may or may not link at the author's discretion: discussions which are very closely bound to a particular API should probably link in the first mention in the given section.

Links between howtos are encouraged. Overview documents and tutorials should always link to reference documents and in depth documents. These documents should link among themselves wherever it's needed: if you're tempted to re-describe the functionality of another module, you should certainly link instead.

Linking to standard library documentation is also encouraged when referencing standard library objects. Intersphinx is supported in Twisted documentation, with prefixes for linking to either the Python 2 standard library documentation (via `py2`) or Python 3 (via `py3`) as needed.

Introductions

The introductory section of a Twisted howto should immediately follow the top-level heading and precede any sub-headings.

The following items should be present in the introduction to Twisted howtos: the introductory paragraph and the description of the target audience.

Introductory paragraph

The introductory paragraph of a document should summarize what the document is designed to present. It should use the both proper names for the Twisted technologies and simple non-Twisted descriptions of the technologies. For example, in this paragraph both the name of the technology ("Conch") and a description ("SSH server") are used:

> This document describes setting up a SSH server to serve data from the file system using Conch, the Twisted SSH implementation.

The introductory paragraph should be relatively short, but should, like the above, somewhere define the document's objective: what the reader should be able to do using instructions in the document.

Description of target audience

Subsequent paragraphs in the introduction should describe the target audience of the document: who would want to read it, and what they should know before they can expect to use your document. For example:

> The target audience of this document is a Twisted user who has a set of filesystem like data objects that they would like to make available to authenticated users over SFTP.
>
> Following the directions in this document will require that you are familiar with managing authentication via the Twisted Cred system.

Use your discretion about the extent to which you list assumed knowledge. Very introductory documents that are going to be among a reader's first exposure to Twisted will even need to specify that they rely on knowledge of Python and of certain networking concepts (ports, servers, clients, connections) but documents that are going to be sought out by existing Twisted users for particular purposes only need to specify other Twisted knowledge that is assumed.

Any knowledge of technologies that wouldn't be considered "core Python" and/or "simple networking" need to be explicitly specified, no matter how obvious they seem to someone familiar with the technology. For example, it needs to be stated that someone using enterprise should know SQL and should know how to set up and populate databases for testing purposes.

Where possible, link to other documents that will fill in missing knowledge for the reader. Linking to documents in the Twisted repository is preferred but not essential.

Goals of document

The introduction should finish with a list of tasks that the user can expect to see the document accomplish. These tasks should be concrete rather than abstract, so rather than telling the user that they will "understand Twisted Conch", you would list the specific tasks that they will see the document do. For example:

> This document will demonstrate the following tasks using Twisted Conch:
>
> - creating an anonymous access read-only SFTP server using a filesystem backend;
> - creating an anonymous access read-only SFTP server using a proxy backend connecting to an HTTP server; and
> - creating an anonymous access read and write SFTP server using a filesystem backend.

In many cases this will essentially be a list of your code examples, but it need not be. If large sections of your code are devoted to design discussions, your goals might resemble the following:

This document will discuss the following design aspects of writing Conch servers:

- authentication of users; and
- choice of data backends.

Example code

Wherever possible, example code should be provided to illustrate a certain technique or piece of functionality.

Example code should try and meet as many of the following requirements as possible:

- example code should be a complete working example suitable for copying and pasting and running by the reader (where possible, provide a link to a file to download);
- example code should be short;
- example code should be commented very extensively, with the assumption that this code may be read by a Twisted newcomer;
- example code should conform to the *coding standard* ; and
- example code should exhibit 'best practice', not only for dealing with the target functionality, but also for use of the application framework and so on.

The requirement to have a complete working example will occasionally impose upon authors the need to have a few dummy functions: in Twisted documentation the most common example is where a function is needed to generate a Deferred and fire it after some time has passed. An example might be this, where deferLater is used to fire a callback after a period of time:

```
from twisted.internet import task, reactor

def getDummyDeferred():
    """
    Dummy method which returns a deferred that will fire in 5 seconds with
    a result
    """
    return task.deferLater(reactor, 5, lambda x: "RESULT")
```

As in the above example, it is imperative to clearly mark that the function is a dummy in as many ways as you can: using `Dummy` in the function name, explaining that it is a dummy in the docstring, and marking particular lines as being required to create an effect for the purposes of demonstration. In most cases, this will save the reader from mistaking this dummy method for an idiom they should use in their Twisted code.

Conclusions

The conclusion of a howto should follow the very last section heading in a file. This heading would usually be called "Conclusion".

The conclusion of a howto should remind the reader of the tasks that they have done while reading the document. For example:

In this document, you have seen how to:

1. set up an anonymous read-only SFTP server;
2. set up a SFTP server where users authenticate;
3. set up a SFTP server where users are restricted to some parts of the filesystem based on authentication; and

4. set up a SFTP server where users have write access to some parts of the filesystem based on authentication.

If appropriate, the howto could follow this description with links to other documents that might be of interest to the reader with their newfound knowledge. However, these links should be limited to fairly obvious extensions of at least one of the listed tasks.

Unit Tests in Twisted

Each *unit test* tests one bit of functionality in the software. Unit tests are entirely automated and complete quickly. Unit tests for the entire system are gathered into one test suite, and may all be run in a single batch. The result of a unit test is simple: either it passes, or it doesn't. All this means you can test the entire system at any time without inconvenience, and quickly see what passes and what fails.

Unit Tests in the Twisted Philosophy

The Twisted development team adheres to the practice of Extreme Programming (XP), and the usage of unit tests is a cornerstone XP practice. Unit tests are a tool to give you increased confidence. You changed an algorithm – did you break something? Run the unit tests. If a test fails, you know where to look, because each test covers only a small amount of code, and you know it has something to do with the changes you just made. If all the tests pass, you're good to go, and you don't need to second-guess yourself or worry that you just accidentally broke someone else's program.

What to Test, What Not to Test

> You don't have to write a test for every single method you write, only production methods that could possibly break.

– Kent Beck, Extreme Programming Explained

Running the Tests

How

From the root of the Twisted source tree, run Trial :

```
$ bin/trial twisted
```

You'll find that having something like this in your emacs init files is quite handy:

```
(defun runtests () (interactive)
  (compile "python /somepath/Twisted/bin/trial /somepath/Twisted"))

(global-set-key [(alt t)] 'runtests)
```

When

Always, always, *always* be sure all the tests pass before committing any code. If someone else checks out code at the start of a development session and finds failing tests, they will not be happy and may decide to *hunt you down* .

Since this is a geographically dispersed team, the person who can help you get your code working probably isn't in the room with you. You may want to share your work in progress over the network, but you want to leave the main Git

tree in good working order. So use a branch , and merge your changes back in only after your problem is solved and all the unit tests pass again.

Adding a Test

Please don't add new modules to Twisted without adding tests for them too. Otherwise we could change something which breaks your module and not find out until later, making it hard to know exactly what the change that broke it was, or until after a release, and nobody wants broken code in a release.

Tests go into dedicated test packages such as `twisted/test/` or `twisted/conch/test/` , and are named `test_foo.py` , where `foo` is the name of the module or package being tested. Extensive documentation on using the PyUnit framework for writing unit tests can be found in the *links section* below.

One deviation from the standard PyUnit documentation: To ensure that any variations in test results are due to variations in the code or environment and not the test process itself, Twisted ships with its own, compatible, testing framework. That just means that when you import the unittest module, you will `from twisted.trial import unittest` instead of the standard `import unittest` .

As long as you have followed the module naming and placement conventions, `trial` will be smart enough to pick up any new tests you write.

PyUnit provides a large number of assertion methods to be used when writing tests. Many of these are redundant. For consistency, Twisted unit tests should use the `assert` forms rather than the `fail` forms. Also, use `assertEqual` , `assertNotEqual` , and `assertAlmostEqual` rather than `assertEquals` , `assertNotEquals` , and `assertAlmostEquals` . `assertTrue` is also preferred over `assert_` . You may notice this convention is not followed everywhere in the Twisted codebase. If you are changing some test code and notice the wrong method being used in nearby code, feel free to adjust it.

When you add a unit test, make sure all methods have docstrings specifying at a high level the intent of the test. That is, a description that users of the method would understand.

Test Implementation Guidelines

Here are some guidelines to follow when writing tests for the Twisted test suite. Many tests predate these guidelines and so do not follow them. When in doubt, follow the guidelines given here, not the example of old unit tests.

Naming Test Classes

When writing tests for the Twisted test suite, test classes are named `FooTests`, where `Foo` is the name of the component being tested. Here is an example:

```
class SSHClientTests(unittest.TestCase):
    def test_sshClient(self):
        foo() # the actual test
```

Real I/O

Most unit tests should avoid performing real, platform-implemented I/O operations. Real I/O is slow, unreliable, and unwieldy.

When implementing a protocol, twisted.test.proto_helpers.StringTransport can be used instead of a real TCP transport. `StringTransport` is fast, deterministic, and can easily be used to exercise all possible network behaviors.

If you need pair a client to a server and have them talk to each other, use twisted.test.iosim.connect with twisted.test.iosim.FakeTransport transports.

Real Time

Most unit tests should also avoid waiting for real time to pass. Unit tests which construct and advance a twisted.internet.task.Clock are fast and deterministic.

When designing your code allow for the reactor to be injected during tests.

```
from twisted.internet.task import Clock

def test_timeBasedFeature(self):
    """
    In this test a Clock scheduler is used.
    """
    clock = Clock()
    yourThing = SomeThing()
    yourThing._reactor = clock

    state = yourThing.getState()

    clock.advance(10)

    # Get state after 10 seconds.
    state = yourThing.getState()
```

Test Data

Keeping test data in the source tree should be avoided where possible.

In some cases it can not be avoided, but where it's obvious how to do so, do it. Test data can be generated at run time or stored inside the test modules as constants.

When file system access is required, dumping data into a temporary path during the test run opens up more testing opportunities. Inside the temporary path you can control various path properties or permissions.

You should design your code so that data can be read from arbitrary input streams.

Tests should be able to run even if they are run inside an installed copy of Twisted.

```
publicRSA_openssh = ("ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAGEArzJx8OYOnJmzf4tf"
"vLi8DVPrJ3/c9k2I/Az64fxjHf9imyRJbixtQhlH9lfNjUIx+4LmrJH5QNRsFporcHDKOTwTT"
"h5KmRpslkYHRivcJSkbh/C+BR3utDS555mV comment")

def test_loadOpenSSHRSAPublic(self):
    """
    L{keys.Key.fromStrea} can load RSA keys serialized in OpenSSH format.
    """
    keys.Key.fromStream(StringIO(publicRSA_openssh))
```

The Global Reactor

Since unit tests are avoiding real I/O and real time, they can usually avoid using a real reactor. The only exceptions to this are unit tests for a real reactor implementation. Unit tests for protocol implementations or other application code

should not use a reactor. Unit tests for real reactor implementations should not use the global reactor, but should instead use twisted.internet.test.reactormixins.ReactorBuilder so they can be applied to all of the reactor implementations automatically. In no case should new unit tests use the global reactor.

Skipping Tests

Trial, the Twisted unit test framework, has some extensions which are designed to encourage developers to add new tests. One common situation is that a test exercises some optional functionality: maybe it depends upon certain external libraries being available, maybe it only works on certain operating systems. The important common factor is that nobody considers these limitations to be a bug.

To make it easy to test as much as possible, some tests may be skipped in certain situations. Individual test cases can raise the `SkipTest` exception to indicate that they should be skipped, and the remainder of the test is not run. In the summary (the very last thing printed, at the bottom of the test output) the test is counted as a"skip" instead of a "success" or "fail" . This should be used inside a conditional which looks for the necessary prerequisites:

```
class SSHClientTests(unittest.TestCase):
    def test_sshClient(self):
        if not ssh_path:
            raise unittest.SkipTest("cannot find ssh, nothing to test")
        foo() # do actual test after the SkipTest
```

You can also set the `.skip` attribute on the method, with a string to indicate why the test is being skipped. This is convenient for temporarily turning off a test case, but it can also be set conditionally (by manipulating the class attributes after they've been defined):

```
class SomeThingTests(unittest.TestCase):
    def test_thing(self):
        dotest()
    test_thing.skip = "disabled locally"
```

```
class MyTests(unittest.TestCase):
    def test_one(self):
        ...
    def test_thing(self):
        dotest()

if not haveThing:
    MyTests.test_thing.im_func.skip = "cannot test without Thing"
    # but test_one() will still run
```

Finally, you can turn off an entire TestCase at once by setting the .skip attribute on the class. If you organize your tests by the functionality they depend upon, this is a convenient way to disable just the tests which cannot be run.

```
class TCPTests(unittest.TestCase):
    ...
class SSLTests(unittest.TestCase):
    if not haveSSL:
        skip = "cannot test without SSL support"
    # but TCPTests will still run
    ...
```

Testing New Functionality

Two good practices which arise from the "XP" development process are sometimes at odds with each other:

- Unit tests are a good thing. Good developers recoil in horror when they see a failing unit test. They should drop everything until the test has been fixed.
- Good developers write the unit tests first. Once tests are done, they write implementation code until the unit tests pass. Then they stop.

These two goals will sometimes conflict. The unit tests that are written first, before any implementation has been done, are certain to fail. We want developers to commit their code frequently, for reliability and to improve coordination between multiple people working on the same problem together. While the code is being written, other developers (those not involved in the new feature) should not have to pay attention to failures in the new code. We should not dilute our well-indoctrinated Failing Test Horror Syndrome by crying wolf when an incomplete module has not yet started passing its unit tests. To do so would either teach the module author to put off writing or committing their unit tests until *after* all the functionality is working, or it would teach the other developers to ignore failing test cases. Both are bad things.

".todo" is intended to solve this problem. When a developer first starts writing the unit tests for functionality that has not yet been implemented, they can set the `.todo` attribute on the test methods that are expected to fail. These methods will still be run, but their failure will not be counted the same as normal failures: they will go into an "expected failures" category. Developers should learn to treat this category as a second-priority queue, behind actual test failures.

As the developer implements the feature, the tests will eventually start passing. This is surprising: after all those tests are marked as being expected to fail. The .todo tests which nevertheless pass are put into a"unexpected success" category. The developer should remove the .todo tag from these tests. At that point, they become normal tests, and their failure is once again cause for immediate action by the entire development team.

The life cycle of a test is thus:

1. Test is created, marked `.todo` . Test fails: "expected failure" .
2. Code is written, test starts to pass. "unexpected success" .
3. `.todo` tag is removed. Test passes. "success" .
4. Code is broken, test starts to fail. "failure" . Developers spring into action.
5. Code is fixed, test passes once more. "success" .

`.todo` may be of use while you are developing a feature, but by the time you are ready to commit anything all the tests you have written should be passing. In other words **never** commit to trunk tests marked as `.todo`. For unfinished tests you should create a follow up ticket and add the tests to the ticket's description.

You can also ignore the `.todo` marker and just make sure you write test first to see them failing before starting to work on the fix.

Line Coverage Information

Trial provides line coverage information, which is very useful to ensure old code has decent coverage. Passing the `--coverage` option to Trial will generate the coverage information in a file called `coverage` which can be found in the `_trial_temp` folder.

Associating Test Cases With Source Files

Please add a `test-case-name` tag to the source file that is covered by your new test. This is a comment at the beginning of the file which looks like one of the following:

```
# -*- test-case-name: twisted.test.test_defer -*-
```

or

```
#!/usr/bin/env python
# -*- test-case-name: twisted.test.test_defer -*-
```

This format is understood by emacs to mark "File Variables" . The intention is to accept `test-case-name` anywhere emacs would on the first or second line of the file (but not in the `File Variables:` block that emacs accepts at the end of the file). If you need to define other emacs file variables, you can either put them in the `File Variables:` block or use a semicolon-separated list of variable definitions:

```
# -*- test-case-name: twisted.test.test_defer; fill-column: 75; -*-
```

If the code is exercised by multiple test cases, those may be marked by using a comma-separated list of tests, as follows: (NOTE: not all tools can handle this yet.. `trial --testmodule` does, though)

```
# -*- test-case-name: twisted.test.test_defer,twisted.test.test_tcp -*-
```

The `test-case-name` tag will allow `trial --testmodule twisted/dir/myfile.py` to determine which test cases need to be run to exercise the code in `myfile.py` . Several tools (as well as http://launchpad.net/twisted-emacs's `twisted-dev.el` 's F9 command) use this to automatically run the right tests.

Links

- A chapter on Unit Testing in Mark Pilgrim's Dive Into Python .
- unittest module documentation, in the Python Library Reference .
- UnitTest on the PortlandPatternRepository Wiki , where all the cool ExtremeProgramming kids hang out.
- Unit Tests in Extreme Programming: A Gentle Introduction .
- Ron Jeffries expounds on the importance of Unit Tests at 100% .
- Ron Jeffries writes about the Unit Test in the Extreme Programming practices of C3 .
- PyUnit's homepage .
- The top-level tests directory, twisted/test.

See also *Tips for writing tests for Twisted code* .

Twisted Compatibility Policy

Motivation

The Twisted project has a small development team, and we cannot afford to provide anything but critical bug-fix support for multiple version branches of Twisted. However, we all want Twisted to provide a positive experience during development, deployment, and usage. Therefore we need to provide the most trouble-free upgrade process possible, so that Twisted application developers will not shy away from upgrades that include necessary bugfixes and feature enhancements.

Twisted is used by a wide variety of applications, many of which are proprietary or otherwise inaccessible to the Twisted development team. Each of these applications is developed against a particular version of Twisted. The most important compatibility to preserve is at the Python API level. Python does not provide us with a strict way to partition **public** and **private** objects (methods, classes, modules), so it is unfortunately quite likely that many of those applications are using arbitrary parts of Twisted. Our compatibility strategy needs to take this into account, and be comprehensive across our entire codebase.

Exceptions can be made for modules aggressively marked **unstable** or **experimental**, but even experimental modules will start being used in production code if they have been around for long enough.

The purpose of this document is to to lay out rules for Twisted application developers who wish to weather the changes when Twisted upgrades, and procedures for Twisted engine developers - both contributors and core team members - to follow when who want to make changes which may be incompatible to Twisted itself.

Defining Compatibility

The word "compatibility" is itself difficult to define. While comprehensive compatibility is good, total compatibility is neither feasible nor desirable. Total compatibility requires that nothing ever change, since any change to Python code is detectable by a sufficiently determined program. There is some folk knowledge around which kind of changes **obviously** won't break other programs, but this knowledge is spotty and inconsistent. Rather than attempt to disallow specific kinds of changes, here we will lay out a list of changes which are considered compatible.

Throughout this document, **compatible** changes are those which meet these specific criteria. Although a change may be broadly considered backward compatible, as long as it does not meet this official standard, it will be officially deemed **incompatible** and put through the process for incompatible changes.

The compatibility policy described here is 99% about changes to **interface**, not changes to functionality.

Note: Ultimately we want to make the user happy but we cannot put every possible thing that will make every possible user happy into this policy.

Brief notes for developers

Here is a summary of the things that need to be done for deprecating code. This is not an exhaustive read and beside this list you should continue reading the rest of this document:

- Do not change the function's behavior as part of the deprecation process.
- Cause imports or usage of the class/function/method to emit a DeprecationWarning either call warnings.warn or use one of the helper APIs
- The warning text must include the version of Twisted in which the function is first deprecated (which will always be a version in the future)
- The warning text should recommend a replacement, if one exists.
- The warning must "point to" the code which called the function. For example, in the normal case, this means stacklevel=2 passed to warnings.warn.
- There must be a unit test which verifies the deprecation warning.
- A .removal news fragment must be added to announce the deprecation.

Procedure for Incompatible Changes

Any change specifically described in the next section as **compatible** may be made at any time, in any release.

The First One's Always Free

The general purpose of this document is to provide a pleasant upgrade experience for Twisted application developers and users.

The specific purpose of this procedure is to achieve that experience by making sure that any application which runs without warnings may be upgraded one minor version of twisted (y to y+1 in x.y.z) or from the last minor revision of a major release to the first minor revision of the next major release (x to x + 1 in x.y.z to x.0.z, when there will be no x.y+1.z).

In other words, any application which runs its tests without triggering any warnings from Twisted should be able to have its Twisted version upgraded at least once with no ill effects except the possible production of new warnings.

Incompatible Changes

Any change which is not specifically described as **compatible** must be made in 2 phases. If a change is made in release R, the timeline is:

1. Release R: New functionality is added and old functionality is deprecated with a DeprecationWarning.
2. At the earliest, release R+2 and one year after release R, but often much later: Old functionality is completely removed.

Removal should happen once the deprecated API becomes an additional maintenance burden.

For example, if it makes implementation of a new feature more difficult, if it makes documentation of non-deprecated APIs more confusing, or if its unit tests become an undue burden on the continuous integration system.

Removal should not be undertaken just to follow a timeline. Twisted should strive, as much as practical, not to break applications relying on it.

Procedure for Exceptions to this Policy

Every change is unique.

Sometimes, we'll want to make a change that fits with this spirit of this document (keeping Twisted working for applications which rely upon it) but may not fit with the letter of the procedure described above (the change modifies behavior of an existing API sufficiently that something might break). Generally, the reason that one would want to do this is to give applications a performance enhancement or bug fix that could break behavior that unanticipated, hypothetical uses of an existing API, but we don't want well-behaved applications to pay the penalty of a deprecation/adopt-a-new-API/removal cycle in order to get the benefits of the improvement if they don't need to.

If this is the case for your change, it's possible to make such a modification without a deprecation/removal cycle. However, we must give users an opportunity to discover whether a particular incompatible change affects them: we should not trust our own assessments of how code uses the API. In order to propose an incompatible change, start a discussion on the mailing list. Make sure that it is eye-catching so those who don't read all list messages in depth will notice it, by prefixing the subject with **INCOMPATIBLE CHANGE:** (capitalized like so). Always include a link to the ticket, and branch (if relevant).

In order to **conclude** such a discussion, there must be a branch available so that developers can run their unit tests against it to mechanically verify that their understanding of their own code is correct. If nobody can produce a failing test or broken application within **a week's time** from such a branch being both 1. available and 2. announced, and at least **three committers** agree that the change is worthwhile, then the branch can be considered approved for the incompatible change in question.

Since some codebases that use Twisted are presumably proprietary and confidential, there should be a good-faith presumption if someone says they have broken tests but cannot immediately produce code to share.

The branch must be available for one week's time.

Note: The announcement forum for incompatible changes and the waiting period required are subject to change as we discover how effective this method is; the important aspect of this policy is that users have some way of finding

out in advance about changes which might affect them.

Compatible Changes. Changed not Covered by the Compatibility Policy

Here is a non-exhaustive list of changes which are not covered by the compatibility policy. These changes can be made without having to worry about the compatibility policy.

Test Changes

No code or data in a test package should be imported or used by a non-test package within Twisted. By doing so, there's no chance anything could access these objects by going through the public API.

Test code and test helpers are considered private API and should not be imported outside of the Twisted testing infrastructure. As an exception to this, twisted.test.proto_helpers is considered a public API (see #6435 for more discussion).

Private Changes

Code is considered *private* if the user would have to type a leading underscore to access it. In other words, a function, module, method, attribute or class whose name begins with an underscore may be arbitrarily changed.

Bug Fixes and Gross Violation of Specifications

If Twisted documents an object as complying with a published specification, and there are inputs which can cause Twisted to behave in obvious violation of that specification, then changes may be made to correct the behavior in the face of those inputs.

If application code must support multiple versions of Twisted, and work around violations of such specifications, then it must test for the presence of such a bug before compensating for it.

For example, Twisted supplies a DOM implementation in twisted.web.microdom. If an issue were discovered where parsing the string *<xml>Hello</xml>* and then serializing it again resulted in *>xml<Hello>/xml<*, that would grossly violate the XML specification for well-formedness. Such code could be fixed with no warning other than release notes detailing that this error is now fixed.

Raw Source Code

The most basic thing that can happen between Twisted versions, of course, is that the code may change. That means that no application may ever rely on, for example, the value of any **func_code** object's **co_code** attribute remaining stable, or the **checksum** of a .py file remaining stable.

Docstrings may also change at any time. No application code may expect any Twisted class, module, or method's __doc__ attribute to remain the same.

New Attributes

New code may also be added. No application may ever rely on the output of the `dir()` function on any object remaining stable, nor on any object's `__all__` attribute, nor on any object's `__dict__` not having new keys added to it. These may happen in any maintenance or bugfix release, no matter how minor.

Pickling

Even though Python objects can be pickled and unpickled without explicit support for this, whether a particular pickled object can be unpickled after any particular change to the implementation of that object is less certain. Because of this, no application may depend on any object defined by Twisted to provide pickle compatibility between any release unless the object explicitly documents this as a feature it has.

Changes Covered by the Compatibility Policy

Here is a non-exhaustive list of changes which are not covered by the compatibility policy.

Some changes appear to be in keeping with the above rules describing what is compatible, but are in fact not.

Interface Changes

Although methods may be added to implementations, adding those methods to interfaces may introduce an unexpected requirement in user code.

Note: There is currently no way to express, in zope.interface, that an interface may optionally provide certain features which need to be tested for. Although we can add new code, we can't add new requirements on user code to implement new methods.

This is easier to deal with in a system which uses abstract base classes because new requirements can provide default implementations which provide warnings. Something could also be put in place to do the same with interfaces, since they already install a metaclass, but this is tricky territory. The only example I'm aware of here is the Microsoft tradition of ISomeInterfaceN where N is a monotonically ascending number for each release.

Private Objects Available via Public Entry Points

If a **public** entry point returns a **private** object, that **private** object must preserve its **public** attributes.

In the following example, `_ProtectedClass` can no longer be arbitrarily changed. Specifically, `getUsers()` is now a public method, thanks to `get_users_database()` exposing it. However, `_checkPassword()` can still be arbitrarily changed or removed.

For example:

```
class _ProtectedClass:
    """
    A private class which is initialized only by an entry point.
    """
    def getUsers(self):
        """
        A public method covered by the compatibility policy.
        """
        return []

    def _checkPassword(self):
        """
        A private method not covered by the compatibility policy.
        """
        return False
```

```
def get_users_database():
    """
    A method guarding the initialization of the private class.

    Since the method is public and it returns an instance of the
    C{_ProtectedClass}, this makes the _ProtectedClass a public class.
    """
    return _ProtectedClass()
```

Private Class Inherited by Public Subclass

A **private** class which is inherited or exposed in any way by **public** subclass will make the inherited class **public**.

The **private** is still protected against direct instantiation.

```
class _Base(object):
    """
    A class which should not be directly instantiated.
    """
    def getActiveUsers(self):
        return []

    def getExpiredusers(self):
        return []

class Users(_Base):
    """
    Public class inheriting from a private class.
    """
    pass
```

In the following example `_Base` is effectively **public**, since `getActiveUsers()` and `getExpiredusers()` are both exposed via the **public** `Users` class.

Documented and Tested Gross Violation of Specifications

If the behaviour of a what was later found as a bug was documented, or fixing it caused existing tests to break, then the change should be considered incompatible, regardless of how gross its violation. It may be that such violations are introduced specifically to deal with other grossly non-compliant implementations of said specification. If it is determined that those reasons are invalid or ought to be exposed through a different API, the change is compatible.

Application Developer Upgrade Procedure

When an application wants to be upgraded to a new version of Twisted, it can do so immediately.

However, if the application wants to get the same **for free** behavior for the next upgrade, the application's tests should be run treating warnings as errors, and fixed.

Supporting and de-supporting Python versions

Twisted does not have a formal policy around supporting new versions of Python or de-supporting old versions of Python. We strive to support Twisted on any version of Python that is the default Python for a vendor-supported release from a major platform, namely Debian, Ubuntu, the latest release of Windows, or the latest release of OS X. The versions of Python currently supported are listed in the INSTALL file for each release.

A distribution release + Python version is only considered supported when a buidlbot builder exists for it.

Removing support for a Python version will be announced at least 1 release prior to the removal.

How to deprecate APIs

Classes

Classes are deprecated by raising a warning when they are access from within their module, using the deprecatedModuleAttribute helper.

```
class SSLContextFactory:
    """
    An SSL context factory.
    """
    deprecatedModuleAttribute(
        Version("Twisted", 12, 2, 0),
        "Use twisted.internet.ssl.DefaultOpenSSLContextFactory instead.",
        "twisted.mail.protocols", "SSLContextFactory")
```

Functions and methods

To deprecate a function or a method, add a call to warnings.warn to the beginning of the implementation of that method. The warning should be of type `DeprecationWarning` and the stack level should be set so that the warning refers to the code which is invoking the deprecated function or method. The deprecation message must include the name of the function which is deprecated, the version of Twisted in which it was first deprecated, and a suggestion for a replacement. If the API provides functionality which it is determined is beyond the scope of Twisted or it has no replacement, then it may be deprecated without a replacement.

There is also a deprecated decorator which works for new-style classes.

For example:

```
import warnings

from twisted.python.deprecate import deprecated
from twisted.python.versions import Version


@deprecated(Version("Twisted", 1, 2, 0), "twisted.baz")
def some_function(bar):
    """
    Function deprecated using a decorator.
    """
    return bar * 3
```

```
@deprecated(Version("Twisted", 1, 2, 0))
def some_function(bar):
    """
    Function deprecated using a decorator and which has no replacement.
    """
    return bar * 3



def some_function(bar):
    """
    Function with a direct call to warnings.
    """
    warnings.warn(
        'some_function is deprecated since Twisted 1.2.0. '
        'Use twisted.baz instead.',
        category=DeprecationWarning,
        stacklevel=2)
    return bar * 3
```

Instance attributes

To deprecate an attribute on instances of a new-type class, make the attribute into a property and call `warnings.warn` from the getter and/or setter function for that property. You can also use the deprecatedProperty decorator which works for new-style classes.

```
from twisted.python.deprecate import deprecated
from twisted.python.versions import Version


class SomeThing(object):
    """
    A class for which the C{user} ivar is not yet deprecated.
    """

    def __init__(self, user):
        self.user = user



class SomeThingWithDeprecation(object):
    """
    A class for which the C{user} ivar is now deprecated.
    """

    def __init__(self, user=None):
        self._user = user


    @deprecatedProperty(Version("Twisted", 1, 2, 0))
    def user(self):
        return self._user


    @user.setter
    def user(self, value):
```

```
        self._user = value
```

Module attributes

Modules cannot have properties, so module attributes should be deprecated using the deprecatedModuleAttribute helper.

```
from twisted.python import _textattributes
from twisted.python.deprecate import deprecatedModuleAttribute
from twisted.python.versions import Version

flatten = _textattributes.flatten

deprecatedModuleAttribute(
    Version('Twisted', 13, 1, 0),
    'Use twisted.conch.insults.text.assembleFormattedText instead.',
    'twisted.conch.insults.text',
    'flatten')
```

Modules

To deprecate an entire module, deprecatedModuleAttribute can be used on the parent package's `__init__.py`.

There are two other options:

- Put a warnings.warn() call into the top-level code of the module.
- Deprecate all of the attributes of the module.

Testing Deprecation Code

Like all changes in Twisted, deprecations must come with associated automated tested. There are several options for checking that a code is deprecated and that using it raises a `DeprecationWarning`.

In order of decreasing preference:

- flushWarnings
- assertWarns
- callDeprecated

```
from twisted.trial import unittest


class DeprecationTests(unittest.TestCase):
    """
    Tests for deprecated code.
    """


    def test_deprecationUsingFlushWarnings(self):
        """
        flushWarnings() is the recommended way of checking for deprecations.
        Make sure you only flushWarning from the targeted code, and not all
```

```
        warnings.
        """
        db.getUser('some-user')

        message = (
            'twisted.Identity.getUser was deprecated in Twisted 15.0.0: '
            'Use twisted.get_user instead.'
            )
        warnings = self.flushWarnings(
            [self.test_deprecationUsingFlushWarnings])
        self.assertEqual(1, len(warnings))
        self.assertEqual(DeprecationWarning, warnings[0]['category'])
        self.assertEqual(message, warnings[0]['message'])


    def test_deprecationUsingAssertWarns(self):
        """
        assertWarns() is designed as a general helper to check any
        type of warnings and can be used for DeprecationsWarnings.
        """
        self.assertWarns(
            DeprecationWarning,
            'twisted.Identity.getUser was deprecated in Twisted 15.0.0 '
            'Use twisted.get_user instead.',
            __file__,
            db.getUser, 'some-user')


    def test_deprecationUsingCallDeprecated(self):
        """
        Avoid using self.callDeprecated() just to check the deprecation
        call.
        """
        self.callDeprecated(
            Version("Twisted", 1, 2, 0), db.getUser, 'some-user')
```

When code is deprecated, all previous tests in which the code is called and tested will now raise `DeprecationWarnings`. Making calls to the deprecated code without raising these warnings can be done using the callDeprecated helper.

```
from twisted.trial import unittest


class IdentityTests(unittest.TestCase):
    """
    Tests for our Identity behavior.
    """

    def test_getUserHomePath(self):
        """
        This is a test in which we check the returned value of C{getUser}
        but we also explicitly handle the deprecations warnings emitted
        during its execution.
        """
        user = self.callDeprecated(
            Version("Twisted", 1, 2, 0), db.getUser, 'some-user')

        self.assertEqual('some-value', user.homePath)
```

Due to a bug in Trial (#6348), unhandled deprecation warnings will not cause test failures or show in test results.

While the Trial bug is not fixed, to trigger test failures on unhandled deprecation warnings use:

```
python -Werror::DeprecationWarning ./bin/trial twisted.conch
```

Working from Twisted's code repository

If you're going to be doing development on Twisted itself, or if you want to take advantage of bleeding-edge features (or bug fixes) that are not yet available in a numbered release, you'll probably want to check out a tree from the Twisted Git repository. The Trunk is where all current development takes place.

This document lists some useful tips for working on this cutting edge.

Checkout

Git tutorials can be found elsewhere, see in particular Git and GitHub learning resources . The relevant data you need to check out a copy of the Twisted tree is available on the development page , and is as follows:

```
$ git clone https://github.com/twisted/twisted Twisted
```

Alternate tree names

By using `git clone https://github.com/twisted/twisted otherdir` , you can put the workspace tree in a directory other than "Twisted" . I do this (with a name like "Twisted-Git") to remind myself that this tree comes from Git and not from a released version (like "Twisted-1.0.5"). This practice can cause a few problems, because there are a few places in the Twisted tree that need to know where the tree starts, so they can add it to `sys.path` without requiring the user manually set their PYTHONPATH. These functions walk the current directory up to the root, looking for a directory named "Twisted" (sometimes exactly that, sometimes with a `.startswith` test). Generally these are test scripts or other administrative tools which expect to be launched from somewhere inside the tree (but not necessarily from the top).

If you rename the tree to something other than `Twisted` , these tools may wind up trying to use Twisted source files from /usr/lib/python2.5 or elsewhere on the default `sys.path` . Normally this won't matter, but it is good to be aware of the issue in case you run into problems.

`twisted/test/process_twisted.py` is one of these programs.

Compiling C extensions

There are currently several C extension modules in Twisted: `twisted.internet.cfsupport` , `twisted.internet.iocpreactor._iocp` , and `twisted.python._epoll` . These modules are optional, but you'll have to compile them if you want to experience their features, performance improvements, or bugs. There are two approaches.

The first is to do a regular distutils `./setup.py build` , which will create a directory under `build/` to hold both the generated `.so` files as well as a copy of the 600-odd `.py` files that make up Twisted. If you do this, you will need to set your PYTHONPATH to something like `MyDir/Twisted/build/lib.linux-i686-2.5` in order to run code against the Git twisted (as opposed to whatever's installed in `/usr/lib/python2.5` or wherever python usually looks). In addition, you will need to re-run the `build` command *every time* you change a `.py` file. The

`build/lib.foo` directory is a copy of the main tree, and that copy is only updated when you re-run `setup.py build`. It is easy to forget this and then wonder why your code changes aren't being expressed.

The second technique is to build the C modules in place, and point your PYTHONPATH at the top of the tree, like `MyDir/Twisted`. This way you're using the .py files in place too, removing the confusion a forgotten rebuild could cause with the separate build/ directory above. To build the C modules in place, do `./setup.py build_ext -i`. You only need to re-run this command when you change the C files. Note that `setup.py` is not Make, it does not always get the dependencies right (`.h` files in particular), so if you are hacking on the cReactor you may need to manually delete the `.o` files before doing a rebuild. Also note that doing a `setup.py clean` will remove the `.o` files but not the final `.so` files, they must be deleted by hand.

Running tests

To run the full unit-test suite, do:

```
./bin/trial twisted
```

To run a single test file (like `twisted/test/test_defer.py`), do one of:

```
./bin/trial twisted.test.test_defer
```

or

```
./bin/trial twisted/test/test_defer.py
```

To run any tests that are related to a code file, like `twisted/protocols/imap4.py`, do:

```
./bin/trial --testmodule twisted/mail/imap4.py
```

This depends upon the `.py` file having an appropriate "test-case-name" tag that indicates which test cases provide coverage. See the *Test Standards* document for details about using "test-case-name". In this example, the `twisted.mail.test.test_imap` test will be run.

Many tests create temporary files in /tmp or ./_trial_temp, but everything in /tmp should be deleted when the test finishes. Sometimes these cleanup calls are commented out by mistake, so if you see a stray `/tmp/@12345.1` directory, it is probably from `test_dirdbm` or `test_popsicle`. Look for an `rmtree` that has been commented out and complain to the last developer who touched that file.

Building docs

Twisted documentation (not including the automatically-generated API docs) is generated by Sphinx. The docs are written in Restructured Text (`.rst`) and translated into `.html` files by the `bin/admin/build-docs` script.

To build the HTML form of the docs into the `doc/` directory, do the following:

```
./bin/admin/build-docs .
```

Committing and Post-commit Hooks

Twisted's Trac installation is notified when the Git repository changes, and will update the ticket depending on the Git commit logs. When making a branch for a ticket, the branch name should end in `-<ticket number>`, for example `my-branch-9999`. This will add a ticket comment containing a changeset link and branch name. To make your commit message show up as a comment on a Trac ticket, add a `refs #<ticket number>` line at

the bottom of your commit message. To automatically close a ticket on Trac as `Fixed` and add a comment with the closing commit message, add a `Fixes: #<ticket number>` line to your commit message. In general, a commit message closing a ticket looks like this:

```
Merge my-branch-9999: A single-line summary.

Author: jesstess
Reviewers: exarkun, glyph
Fixes: #9999

My longer description of the changes made.
```

The *Twisted Coding Standard* elaborates on commit messages and source control.

Emacs

A minor mode for development with Twisted using Emacs is available. See `twisted-dev.el` , provided by twisted-emacs , for several utility functions which make it easier to grep for methods, run test cases, etc.

Building Debian packages

Our support for building Debian packages has fallen into disrepair. We would very much like to restore this functionality, but until we do so, if you are interested in this, you are on your own. See stdeb for one possible approach to this.

Travis CI integration

Travis CI is configured to run a subset of the official Buildbot builders for each push to a PR or to `trunk`

The tests are executed using `tox-travis`. See the `tox.ini` file for the actual configuration.

Twisted Release Process

This document describes the Twisted release process. Although it is still incomplete, every effort has been made to ensure that it is accurate and up-to-date.

This process has only been tested on Linux or OS X, so we recommend that you do the release on Linux or OS X.

If you want to make changes to the release process, follow the normal Twisted development process (contribute release automation software that has documentation and unit tests demonstrating that it works).

Outcomes

By the end of a Twisted release we'll have:

- Tarballs for Twisted as a whole, and for each of its sub-projects
- Windows installers for the whole Twisted project
- Updated documentation (API & howtos) on the twistedmatrix.com site
- Updated documentation on Read The Docs
- Updated download links on the twistedmatrix.com site

- Announcement emails sent to major Python lists
- Announcement post on the Twisted blog
- A tag in our Git repository marking the release

Prerequisites

To release Twisted, you will need:

- Commit privileges to Twisted
- Access to `dornkirk.twistedmatrix.com` as t-web
- Permissions to edit the Downloads wiki page
- Channel operator permissions for `#twisted`
- Admin privileges for Twisted's PyPI packages
- Contributor status for the Twisted blog
- Read The Docs access for the Twisted project

Version numbers

Twisted releases use a time-based numbering scheme. Releases versions like YY.MM.mm, where YY is the last two digits of the year of the release, MM is the month of release, and mm is the number of the patch release.

For example:

- A release in Jan 2017 is 17.1.0
- A release in Nov 2017 is 17.11.0
- If 17.11.0 has some critical defects, then a patch release would be numbered 17.11.1
- The first release candidate of 17.1.0 is 17.1.0rc1, the second is 17.1.0rc2

Every release of Twisted includes the whole project.

Throughout this document, we'll refer to the version number of the release as $RELEASE. Examples of $RELEASE include 10.0.0, 10.1.0, 10.1.1 etc.

We'll refer to the first two components of the release as $API, since all releases that share those numbers are mutually API compatible. e.g. for 10.0.0, $API is 10.0; for 10.1.0 and 10.1.1, $API is 10.1.

Incremental automatically picks the correct version number for you. Please retrieve it after you run it.

Overview

To release Twisted, we

1. Prepare for a release
2. Release one or more release candidates
3. Release the final release

Prepare for a release

1. Check the milestone for the upcoming release
   - Get rid of any non-critical bugs
   - Get any critical bugs fixed
   - Check the release manager notes in case anyone has left anything which can only be done during the release.
2. Check for any regressions
3. Read through the `INSTALL.rst` and `README.rst` files to make sure things like the supported Python versions are correct
   - Check the required Python version.
   - Check that the list matches the current set of buildbots.
   - Any mistakes should be fixed in trunk before making the release branch
4. Choose a version number.
5. File a ticket
   - Assign it to the upcoming release milestone
   - Assign it to yourself
   - Call it "Release $RELEASE"
6. Make a branch and attach it to the ticket:
   - `git fetch origin`
   - `git checkout origin/trunk`
   - `git checkout -b release-$RELEASE-4290`

How to do a release candidate

1. Check buildbot to make sure all supported platforms are green (wait for pending builds if necessary).
2. If a previously supported platform does not currently have a buildbot, move from supported platforms to "expected to work" in `INSTALL.rst`.
3. In your Git repo, fetch and check out the new release branch.
4. Run `python -m incremental.update Twisted --rc`
5. Commit the changes made by Incremental.
6. Run `towncrier`.
7. Commit the changes made by towncrier - this automatically removes the NEWS newsfragments.
8. Bump copyright dates in `LICENSE`, `twisted/copyright.py`, and `README.rst` if required
9. Push the changes up to GitHub.
10. Run `python setup.py sdist --formats=bztar -d /tmp/twisted-release` to build the tarballs.
11. Copy `NEWS.rst` to `/tmp/twisted-release/` for people to view without having to download the tarballs. (e.g. `cp NEWS.rst /tmp/twisted-release/NEWS.rst`)
12. Upload the tarballs to `twistedmatrix.com/Releases/rc/$RELEASE` (see #4353)

- You can use `rsync --rsh=ssh --partial --progress -av /tmp/twisted-release/ t-web@dornkirk.twistedmatrix.com:/srv/t-web/data/releases/rc/<RELEASE>/` to do this.

13. Write the release candidate announcement
    - Read through the NEWS file and summarize the interesting changes for the release
    - Get someone else to look over the announcement before doing it
14. Announce the release candidate on
    - the twisted-python mailing list
    - on IRC in the `#twisted` topic

Release candidate announcement

The release candidate announcement should mention the important changes since the last release, and exhort readers to test this release candidate.

Here's what the $RELEASErc1 release announcement might look like:

```
Live from PyCon Atlanta, I'm pleased to herald the approaching
footsteps of the $API release.

Tarballs for the first Twisted $RELEASE release candidate are now available at:
 http://people.canonical.com/~jml/Twisted/

Highlights include:

 * Improved documentation, including "Twisted Web in 60 seconds"

 * Faster Perspective Broker applications

 * A new Windows installer that ships without zope.interface

 * Twisted no longer supports Python 2.3

 * Over one hundred closed tickets

For more information, see the NEWS file.

Please download the tarballs and test them as much as possible.

Thanks,
jml
```

A week is a generally good length of time to wait before doing the final release.

How to do a final release

Prepare the branch

1. Have the release branch, previously used to generate a release candidate, checked out
2. Run `python -m incremental.update Twisted`.

3. Revert the release candidate newsfile changes, in order.
4. Run `towncrier` to make the final newsfile.
5. Add the quote of the release to the `README.rst`
6. Make a new quote file for the next version
   - `git mv docs/fun/Twisted.Quotes docs/historic/Quotes/Twisted-$API; echo '' > docs/fun/Twisted.Quotes; git add docs/fun/Twisted.Quotes`
7. Commit the version and `README.rst` changes.
8. Submit the ticket for review
9. Pause until the ticket is reviewed and accepted.
10. Tag the release.

- `git tag -s twisted-$RELEASE -m "Tag $RELEASE release"`
- `git push --tags`

Cut the tarballs & installers

1. Using a checkout of the release branch or the release tag (with no local changes!), build the tarballs:
   - `python setup.py sdist --formats=bztar -d /tmp/twisted-release`
2. Build Windows wheel

- Download the latest `.whl` files from Buildbot and save them in the staging directory

3. Sign the tarballs and Windows installers. (You will need a PGP key for this - use something like Seahorse to generate one, if you don't have one.)

- MD5: `md5sum Tw* | gpg -a --clearsign > /tmp/twisted-release/twisted-$RELEASE-md5sums.txt`
- SHA512: `shasum -a 512 Tw* | gpg -a --clearsign > /tmp/twisted-release/twisted-$RELEASE-shasums.txt`
- Compare these to an example of `twisted-$RELEASE-md5sums.txt` - they should look the same.

Update documentation

1. Get the dependencies

- PyDoctor (from PyPI)

2. Build the documentation

- `./bin/admin/build-docs .`
- `cp -R doc /tmp/twisted-release/`

3. Run the build-apidocs script to build the API docs and then upload them (See also #2891).

- Copy the pydoctor directory from the twisted branch into your Git checkout.
- `./bin/admin/build-apidocs . /tmp/twisted-release/api`
- Documentation will be generated in a directory called `/tmp/twisted-release/api`

4. Update the Read The Docs default to point to the release branch (via the dashboard).

Distribute

1. Create a tarball with the contents of the release directory: `cd /tmp/twisted-release; tar -cvjf ../release.tar.bz2 *`
2. Upload to the official upload locations (see #2888)

- `cd ~; git clone https://github.com/twisted-infra/braid`
- `cd braid`
- `virtualenv ~/dev/braid; source ~/dev/braid/bin/activate; cd ~/braid; python setup.py develop;`
- `cd ~/braid; fab config.production t-web.uploadRelease:$RELEASE,/tmp/release.tar.bz2`

3. Test the generated docs

- Browse to `http://twistedmatrix.com/documents/$RELEASE/`
- Make sure that there is content in each of the directories and that it looks good
- Follow each link on the documentation page, replace current with `$RELEASE` (e.g. 10.0.0) and look for any obvious breakage

4. Change the "current" symlink
    - Upload release: `fab config.production t-web.updateCurrentDocumentation:$RELEASE`

Announce

1. Update Downloads pages

- The following updates are automatic, due to the use of the ProjectVersion wiki macro throughout most of the Downloads page.
    - Text references to the old version to refer to the new version
    - The link to the NEWS file to point to the new version
    - Links and text to the main tarball
- Add a new md5sum link
- Add a new shasum link
- Save the page, check all links

2. Update PyPI records & upload files
    - `pip install -U twine`
    - `twine upload /tmp/twisted-release/Twisted-$RELEASE*`
3. Write the release announcement (see below)
4. Announce the release

- Send a text version of the announcement to: twisted-python@twistedmatrix.com, python-announce-list@python.org, python-list@python.org, twisted-web@twistedmatrix.com
- http://labs.twistedmatrix.com (Post a web version of the announcements, with links instead of literal URLs)
- Twitter, if you feel like it

- `#twisted` topic on IRC (you'll need ops)

5. Run `python -m incremental Twisted --dev` to add a *dev0* postfix.
6. Commit the dev0 update change.
7. Merge the release branch into trunk, closing the release ticket at the same time.
8. Close the release milestone (which should have no tickets in it).
9. Open a milestone for the next release.

Release announcement

The final release announcement should:

- Mention the version number
- Include links to where the release can be downloaded
- Summarize the significant changes in the release
- Consider including the quote of the release
- Thank the contributors to the release

Here's an example:

```
On behalf of Twisted Matrix Laboratories, I am honoured to announce
the release of Twisted 13.2!

The highlights of this release are:

 * Twisted now includes a HostnameEndpoint implementation which uses
IPv4 and IPv6 in parallel, speeding up the connection by using
whichever connects first (the 'Happy Eyeballs'/RFC 6555 algorithm).
(#4859)

 * Improved support for Cancellable Deferreds by kaizhang, our GSoC
student. (#4320, #6532, #6572, #6639)

 * Improved Twisted.Mail documentation by shira, our Outreach Program
for Women intern. (#6649, #6652)

 * twistd now waits for the application to start successfully before
exiting after daemonization. (#823)

 * SSL server endpoint string descriptions now support the
specification of chain certificates. (#6499)

 * Over 70 closed tickets since 13.1.0.

For more information, check the NEWS file (link provided below).

You can find the downloads at <https://pypi.python.org/pypi/Twisted>
(or alternatively <http://twistedmatrix.com/trac/wiki/Downloads>) .
The NEWS file is also available at
<http://twistedmatrix.com/Releases/Twisted/13.2/NEWS.txt>.

Many thanks to everyone who had a part in this release - the
supporters of the Twisted Software Foundation, the developers who
```

```
contributed code as well as documentation, and all the people building
great things with Twisted!

Twisted Regards,
HawkOwl
```

When things go wrong

If you discover a showstopper bug during the release process, you have three options.

1. Abort the release, make a new point release (e.g. abort 10.0.0, make 10.0.1 after the bug is fixed)
2. Abort the release, make a new release candidate (e.g. abort 10.0.0, make 10.0.0pre3 after the bug is fixed)
3. Interrupt the release, fix the bug, then continue with it (e.g. release 10.0.0 with the bug fix)

If you choose the third option, then you should:

- Delete the tag for the release
- Recreate the tag from the release branch once the fix has been applied to that branch

Bug fix releases

Sometimes, bugs happen, and sometimes these are regressions in the current released version. This section goes over doing these "point" releases.

1. Ensure all bugfixes are in trunk.
2. Make a branch off the affected version.
3. Cherry-pick the merge commits that merge the bugfixes into trunk, onto the new release branch.
4. Go through the rest of the process for a full release from "How to do a release candidate", merging the release branch into trunk as normal as the end of the process.

- Instead of just `--rc` when running the change-versions script, add the patch flag, making it `--patch --rc`.
- Instead of waiting a week, a shorter pause is acceptable for a patch release.

Open questions

- How do we manage the case where there are untested builds in trunk?
- Should picking a release quote be part of the release or the release candidate?
- What bugs should be considered release blockers?
  - All bugs with a type from the release blocker family
  - Anybody can create/submit a new ticket with a release blocker type
  - Ultimately it's the RM's discretion to accept a ticket as a release blocker
- Should news fragments contain information about who made the changes?

This series of documents is designed for people who wish to contribute to the Twisted codebase.

- *Coding standard*
- *Documentation writing standard*

- *Testing standard*
- *Compatibility Policy*
- *Working from Twisted's Git repository*
- *Releasing Twisted*

This documentation is for people who work on the Twisted codebase itself, rather than for people who want to use Twisted in their own projects.

- *Naming*
- *Philosophy*
- *Security*
- *Twisted development policy*
- *Developer guides* : documentation on using Twisted Core to develop your own applications
- *Examples* : short code examples using Twisted Core
- *Specifications* : specification documents for elements of Twisted Core
- *Development of Twisted* : for people who want to work on Twisted itself

An API reference is available on the twistedmatrix web site.

CHAPTER 3

Twisted Conch

Developer Guides

Writing a client with Twisted Conch

Introduction

In the original days of computing, rsh/rlogin were used to connect to remote computers and execute commands. These commands had the problem that the passwords and commands were sent in the clear. To solve this problem, the SSH protocol was created. Twisted Conch implements the second version of this protocol.

Using an SSH Command Endpoint

If your objective is to execute a command on a remote host over an SSH connection, then the easiest approach may be to use twisted.conch.endpoints.SSHCommandClientEndpoint . If you haven't used endpoints before, first take a look at *the endpoint howto* to get an idea of how endpoints work in general.

Conch provides an endpoint implementation which establishes an SSH connection, performs necessary authentication, opens a channel, and launches a command in that channel. It then associates the output of that command with the input of a protocol you supply, and associates output from that protocol with the input of that command. Effectively, this lets you ignore most of the complexity of SSH and just interact with a remote process as though it were any other stream-oriented connection - such as TCP or SSL.

Conch also provides an endpoint that is initialized with an already established SSH connection. This endpoint just opens a new channel on the existing connection and launches a command in that.

Using the `SSHCommandClientEndpoint` is about as simple as using any other stream-oriented client endpoint. Just create the endpoint defining where the SSH server to connect to is and a factory defining what kind of protocol to use to interact with the command and let them get to work using the endpoint's `connect` method.

`echoclient_ssh.py`

```
#!/usr/bin/env python
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.
from __future__ import print_function

if __name__ == '__main__':
    import sys
    import echoclient_ssh
    from twisted.internet.task import react
    react(echoclient_ssh.main, sys.argv[1:])

import os, getpass

from twisted.python.filepath import FilePath
from twisted.python.usage import Options
from twisted.internet.defer import Deferred
from twisted.internet.protocol import Factory, Protocol
from twisted.internet.endpoints import UNIXClientEndpoint
from twisted.conch.ssh.keys import EncryptedKeyError, Key
from twisted.conch.client.knownhosts import KnownHostsFile
from twisted.conch.endpoints import SSHCommandClientEndpoint


class EchoOptions(Options):
    optParameters = [
        ("host", "h", "localhost",
         "hostname of the SSH server to which to connect"),
        ("port", "p", 22,
         "port number of SSH server to which to connect", int),
        ("username", "u", getpass.getuser(),
         "username with which to authenticate with the SSH server"),
        ("identity", "i", None,
         "file from which to read a private key to use for authentication"),
        ("password", None, None,
         "password to use for authentication"),
        ("knownhosts", "k", "~/.ssh/known_hosts",
         "file containing known ssh server public key data"),
        ]

    optFlags = [
        ["no-agent", None, "Disable use of key agent"],
        ]



class NoiseProtocol(Protocol):
    def connectionMade(self):
        self.finished = Deferred()
        self.strings = ["bif", "pow", "zot"]
        self.sendNoise()


    def sendNoise(self):
        if self.strings:
            self.transport.write(self.strings.pop(0) + "\n")
        else:
            self.transport.loseConnection()
```

```
    def dataReceived(self, data):
        print("Server says:", data)
        self.sendNoise()


    def connectionLost(self, reason):
        self.finished.callback(None)



def readKey(path):
    try:
        return Key.fromFile(path)
    except EncryptedKeyError:
        passphrase = getpass.getpass("%r keyphrase: " % (path,))
        return Key.fromFile(path, passphrase=passphrase)



class ConnectionParameters(object):
    def __init__(self, reactor, host, port, username, password, keys,
                 knownHosts, agent):
        self.reactor = reactor
        self.host = host
        self.port = port
        self.username = username
        self.password = password
        self.keys = keys
        self.knownHosts = knownHosts
        self.agent = agent


    @classmethod
    def fromCommandLine(cls, reactor, argv):
        config = EchoOptions()
        config.parseOptions(argv)

        keys = []
        if config["identity"]:
            keyPath = os.path.expanduser(config["identity"])
            if os.path.exists(keyPath):
                keys.append(readKey(keyPath))

        knownHostsPath = FilePath(os.path.expanduser(config["knownhosts"]))
        if knownHostsPath.exists():
            knownHosts = KnownHostsFile.fromPath(knownHostsPath)
        else:
            knownHosts = None

        if config["no-agent"] or "SSH_AUTH_SOCK" not in os.environ:
            agentEndpoint = None
        else:
            agentEndpoint = UNIXClientEndpoint(
                reactor, os.environ["SSH_AUTH_SOCK"])

        return cls(
            reactor, config["host"], config["port"],
```

```
            config["username"], config["password"], keys,
            knownHosts, agentEndpoint)


    def endpointForCommand(self, command):
        return SSHCommandClientEndpoint.newConnection(
            self.reactor, command, self.username, self.host,
            port=self.port, keys=self.keys, password=self.password,
            agentEndpoint=self.agent, knownHosts=self.knownHosts)



def main(reactor, *argv):
    parameters = ConnectionParameters.fromCommandLine(reactor, argv)
    endpoint = parameters.endpointForCommand(b"/bin/cat")

    factory = Factory()
    factory.protocol = NoiseProtocol

    d = endpoint.connect(factory)
    d.addCallback(lambda proto: proto.finished)
    return d
```

For completeness, this example includes a lot of code to support different styles of authentication, reading (and possibly updating) existing *known_hosts* files, and parsing command line options. Focus on the latter half of the `main` function to see the code that is most directly responsible for actually doing the necessary SSH connection setup. `SSHCommandClientEndpoint` accepts quite a few options, since there is a lot of flexibility in SSH and many possible different server configurations, but once the endpoint object itself is created, its use is no more complicated than the use of any other endpoint: pass a factory to its `connect` method and attach a callback to the resulting `Deferred` to do something with the protocol instance. If you use an endpoint that creates new connections, the connection attempt can be cancelled by calling `cancel()` on this `Deferred`.

In this case, the connected protocol instance is only used to make the example wait until the client has finished talking to the server, which happens after the small amount of example data has been sent to the server and bounced back by the `/bin/cat` process the protocol is interacting with.

Several of the options accepted by `SSHCommandClientEndpoint.newConnection` should be easy to understand. The endpoint takes a reactor which it uses to do any and all I/O it needs to do. It also takes a command which it executes on the remote server once the SSH connection is established and authenticated; this command is a single string, perhaps including spaces or other special shell symbols, and is interpreted by a shell on the server. It takes a username with which it identifies itself to the server for authentication purposes. It takes an optional password argument which will also be used for authentication - if the server supports password authentication (prefer keys instead where possible, see below). It takes a host (either a name or an IP address) and a port number, defining where to connect.

Some of the other options may bear further explanation.

The `keys` argument gives any SSH Key objects which may be useful for authentication. These keys are available to the endpoint for authentication, but only keys that the server indicates are useful will actually be used. This argument is optional. If key authentication against the server is either unnecessary or undesired, it may be omitted entirely.

The `agentEndpoint` argument gives the `SSHCommandClientEndpoint` an opportunity to connect to an SSH authentication agent. The agent may already be loaded with keys, or may have some other way to authenticate a connection. Using the agent can mean the process actually establishing the SSH connection doesn't need to load any authentication material (passwords or keys) itself (often convenient in case keys are encrypted and potentially more secure, since only the agent process ever actually holds the secrets). The value for this argument is another `IStreamClientEndpoint`. Often in a typical *NIX desktop environment, the *SSH_AUTH_SOCK* environment

variable will give the location of an AF_UNIX socket. This explains the value `echoclient_ssh.py` assigns this parameter when *–no-agent* is not given.

The `knownHosts` argument accepts a KnownHostsFile instance and controls how server keys are checked and stored. This object has the opportunity to reject server keys if they differ from expectations. It can also save server keys when they are first observed.

Finally, there is one option that is not demonstrated in the example - the `ui` argument. This argument is closely related to the `knownHosts` argument described above. `KnownHostsFile` may require user-input under certain circumstances - for example, to ask if it should accept a server key the first time it is observed. The `ui` object is how this user-input is obtained. By default, a ConsoleUI instance associated with */dev/tty* will be used. This gives about the same behavior as is seen in a standard command-line ssh client. See SSHCommandClientEndpoint.newConnection for details about how edge cases are handled for this default value. For use of `SSHCommandClientEndpoint` that is intended to be completely autonomous, applications will probably want to specify a custom `ui` object which can make the necessary decisions without user-input.

It is also possible to run commands (one or more) over an already-established connection. This is done using the alternate constructor `SSHCommandClientEndpoint.existingConnection`. The `connection` argument to that function can be obtained by accessing `transport.conn` on an already connected protocol.

`echoclient_shared_ssh.py`

```
#!/usr/bin/env python
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.
from __future__ import print_function

if __name__ == '__main__':
    import sys
    import echoclient_shared_ssh
    from twisted.internet.task import react
    react(echoclient_shared_ssh.main, sys.argv[1:])

from twisted.internet.task import cooperate
from twisted.internet.defer import Deferred, gatherResults
from twisted.internet.protocol import Factory, Protocol

from twisted.conch.endpoints import SSHCommandClientEndpoint

from echoclient_ssh import ConnectionParameters


class PrinterProtocol(Protocol):
    def dataReceived(self, data):
        print("Got some data:", data, end=' ')


    def connectionLost(self, reason):
        print("Lost my connection")
        self.factory.done.callback(None)



def main(reactor, *argv):
    parameters = ConnectionParameters.fromCommandLine(reactor, argv)
    endpoint = parameters.endpointForCommand(b"/bin/cat")

    done = []
    factory = Factory()
```

```
    factory.protocol = Protocol
    d = endpoint.connect(factory)

    def gotConnection(proto):
        conn = proto.transport.conn

        for i in range(50):
            factory = Factory()
            factory.protocol = PrinterProtocol
            factory.done = Deferred()
            done.append(factory.done)

            e = SSHCommandClientEndpoint.existingConnection(
                conn, b"/bin/echo %d" % (i,))
            yield e.connect(factory)

    d.addCallback(gotConnection)
    d.addCallback(lambda work: cooperate(work).whenDone())
    d.addCallback(lambda ignored: gatherResults(done))

    return d
```

Writing a client

In case the endpoint is missing some necessary functionality, or in case you want to interact with a different part of an SSH server - such as one of its *subsystems* (for example, SFTP), you may need to use the lower-level Conch client interface. This is described below.

Writing a client with Conch involves sub-classing 4 classes: twisted.conch.ssh.transport.SSHClientTransport , twisted.conch.ssh.userauth.SSHUserAuthClient , twisted.conch.ssh.connection.SSHConnection , and twisted.conch.ssh.channel.SSHChannel . We'll start out with `SSHClientTransport` because it's the base of the client.

The Transport

```
from twisted.conch import error
from twisted.conch.ssh import transport
from twisted.internet import defer

class ClientTransport(transport.SSHClientTransport):

    def verifyHostKey(self, pubKey, fingerprint):
        if fingerprint != 'b1:94:6a:c9:24:92:d2:34:7c:62:35:b4:d2:61:11:84':
            return defer.fail(error.ConchError('bad key'))
        else:
            return defer.succeed(1)

    def connectionSecure(self):
        self.requestService(ClientUserAuth('user', ClientConnection()))
```

See how easy it is? `SSHClientTransport` handles the negotiation of encryption and the verification of keys for you. The one security element that you as a client writer need to implement is `verifyHostKey()` . This method is called with two strings: the public key sent by the server and its fingerprint. You should verify the host key the server sends, either by checking against a hard-coded value as in the example, or by asking the user. `verifyHostKey` returns a twisted.internet.defer.Deferred which gets a callback if the host key is valid, or an errback if it is not. Note that

in the above, replace 'user' with the username you're attempting to ssh with, for instance a call to `os.getlogin()` for the current user.

The second method you need to implement is `connectionSecure()` . It is called when the encryption is set up and other services can be run. The example requests that the `ClientUserAuth` service be started. This service will be discussed next.

The Authorization Client

```
from twisted.conch.ssh import keys, userauth

# these are the public/private keys from test_conch

publicKey = 'ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAGEArzJx8OYOnJmzf4tfBEvLi8DVPrJ3\
/c9k2I/Az64fxjHf9imyRJbixtQhlH9lfNjUIx+4LmrJH5QNRsFporcHDKOTwTTYLh5KmRpslkYHR\
ivcJSkbh/C+BR3utDS555mV'

privateKey = """-----BEGIN RSA PRIVATE KEY-----
MIIByAIBAAJhAK8ycfDmDpyZs3+LXwRLy4vA1T6yd/3PZNiPwM+uH8Yx3/YpskSW
4sbUIZR/ZXzY1CMfuC5qyR+UDUbBaaK3Bwyjk8E02C4eSpkabJZGB0Yr3CUpG4fw
vgUd7rQ0ueeZlQIBIwJgbh+1VZfr7WftK5lu7MHtqE1S1vPWZQYE3+VUn8yJADyb
Z4fsZaCrzW9lkIqXkE3GIY+ojdhZhkO1gbG0118sIgphwSWKRxK0mvh6ERxKqIt1
xJEJO74EykXZV4oNJ8sjAjEA3J9r2ZghVhGN6V8DnQrTk24Td0E8hU8AcP0FVP+8
PQm/g/aXf2QQkQT+omdHVEJrAjEAy0pL0EBH6EVS98evDCBtQw22OZT52qXlAwZ2
gyTriKFVoqjeEjt3SZKKqXHSApP/AjBLpF99zcJJZRq2abgYlf9lv1chkrWqDHUu
DZttmYJeEfiFBBavVYIF1dOlZT0G8jMCMBc7sOSZodFnAiryP+Qg9otSBjJ3bQML
pSTqy7c3a2AScC/YyOwkDaICHnnD3XyjMwIxALRzl0tQEKMXs6hH8ToUdlLROCrP
EhQ0wahUTCk1gKA4uPD6TMTChavbh4K63OvbKg==
-----END RSA PRIVATE KEY-----"""

class ClientUserAuth(userauth.SSHUserAuthClient):

    def getPassword(self, prompt = None):
        return
        # this says we won't do password authentication

    def getPublicKey(self):
        return keys.Key.fromString(data = publicKey).blob()

    def getPrivateKey(self):
        return defer.succeed(keys.Key.fromString(data = privateKey).keyObject)
```

Again, fairly simple. The `SSHUserAuthClient` takes care of most of the work, but the actual authentication data needs to be supplied. `getPassword()` asks for a password, `getPublicKey()` and `getPrivateKey()` get public and private keys, respectively. `getPassword()` returns a `Deferred` that is called back with the password to use.

`getPublicKey()` returns the SSH key data for the public key to use. twisted.conch.ssh.keys.Key.fromString() will take a key in OpenSSH, LSH or any supported format, as a string, and generate a new twisted.conch.ssh.keys.Key. Alternatively, `keys.Key.fromFile()` can be used instead, which will take the filename of a key in the supported format, and and generate a new twisted.conch.ssh.keys.Key.

`getPrivateKey()` returns a `Deferred` which is called back with the private twisted.conch.ssh.keys.Key.

`getPassword()` and `getPrivateKey()` return `Deferreds` because they may need to ask the user for input.

Once the authentication is complete, `SSHUserAuthClient` takes care of starting the code `SSHConnection` object given to it. Next, we'll look at how to use the `SSHConnection`

The Connection

```
from twisted.conch.ssh import connection

class ClientConnection(connection.SSHConnection):

    def serviceStarted(self):
        self.openChannel(CatChannel(conn = self))
```

`SSHConnection` is the easiest, as it's only responsible for starting the channels. It has other methods, those will be examined when we look at `SSHChannel` .

The Channel

```
from twisted.conch.ssh import channel, common

class CatChannel(channel.SSHChannel):

    name = 'session'

    def channelOpen(self, data):
        d = self.conn.sendRequest(self, 'exec', common.NS('cat'),
                                  wantReply = 1)
        d.addCallback(self._cbSendRequest)
        self.catData = ''

    def _cbSendRequest(self, ignored):
        self.write('This data will be echoed back to us by "cat."\r\n')
        self.conn.sendEOF(self)
        self.loseConnection()

    def dataReceived(self, data):
        self.catData += data

    def closed(self):
        print('We got this from "cat":', self.catData)
```

Now that we've spent all this time getting the server and client connected, here is where that work pays off. `SSHChannel` is the interface between you and the other side. This particular channel opens a session and plays with the 'cat' program, but your channel can implement anything, so long as the server supports it.

The `channelOpen()` method is where everything gets started. It gets passed a chunk of data; however, this chunk is usually nothing and can be ignored. Our `channelOpen()` initializes our channel, and sends a request to the other side, using the `sendRequest()` method of the `SSHConnection` object. Requests are used to send events to the other side. We pass the method self so that it knows to send the request for this channel. The 2nd argument of 'exec' tells the server that we want to execute a command. The third argument is the data that accompanies the request. common.NS encodes the data as a length-prefixed string, which is how the server expects the data. We also say that we want a reply saying that the process has a been started. `sendRequest()` then returns a `Deferred` which we add a callback for.

Once the callback fires, we send the data. `SSHChannel` supports the twisted.internet.interfaces.ITransport interface, so it can be given to Protocols to run them over the secure connection. In our case, we just write the data directly. `sendEOF()` does not follow the interface, but Conch uses it to tell the other side that we will write no more data. `loseConnection()` shuts down our side of the connection, but we will still receive data through `dataReceived()` . The `closed()` method is called when both sides of the connection are closed, and we use it to display the data we received (which should be the same as the data we sent.)

Finally, let's actually invoke the code we've set up.

The main() function

```
from twisted.internet import protocol, reactor

def main():
    factory = protocol.ClientFactory()
    factory.protocol = ClientTransport
    reactor.connectTCP('localhost', 22, factory)
    reactor.run()

if __name__ == "__main__":
    main()
```

We call `connectTCP()` to connect to localhost, port 22 (the standard port for ssh), and pass it an instance of twisted.internet.protocol.ClientFactory . This instance has the attribute `protocol` set to our earlier `ClientTransport` class. Note that the protocol attribute is set to the class `ClientTransport` , not an instance of `ClientTransport` ! When the `connectTCP` call completes, the protocol will be called to create a `ClientTransport()` object - this then invokes all our previous work.

It's worth noting that in the example `main()` routine, the `reactor.run()` call never returns. If you want to make the program exit, call `reactor.stop()` in the earlier `closed()` method.

If you wish to observe the interactions in more detail, adding a call to `log.startLogging(sys.stdout, setStdout=0)` before the `reactor.run()` call will send all logging to stdout.

- Tutorial
  - *Writing an SSH client with Conch*

Examples

Simple SSH server and client

- `sshsimpleclient.py` - simple SSH client
- `sshsimpleserver.py` - simple SSH server

Simple telnet server

- `telnet_echo.tac` - A telnet server which echoes data and events back to the client

twisted.conch.insults examples

- `demo.tac` - Nearly pointless demonstration of the manhole interactive interpreter
- `demo_draw.tac` - A trivial drawing application
- `demo_insults.tac` - Various simple terminal manipulations using the insults module
- `demo_recvline.tac` - Demonstrates line-at-a-time handling with basic line-editing support
- `demo_scroll.tac` - Simple echo-ish server that uses the scroll-region

- `demo_manhole.tac` - An interactive Python interpreter with syntax coloring
- `window.tac` - An example of various widgets
- *Developer guides*: documentation on using Twisted Conch to develop your own applications
- *Examples*: short code examples using Twisted Conch

CHAPTER 4

Twisted Mail

Examples

SMTP servers

- `emailserver.tac` - a toy email server.

SMTP clients

- `sendmail_smtp.py` - sending email over plain SMTP with the high-level sendmail client.
- `sendmail_gmail.py` - sending email encrypted ESMTP to GMail with the high-level sendmail client.
- `sendmail_message.py` - sending a complex message with the high-level sendmail client.
- `smtpclient_simple.py` - sending email using SMTP.
- `smtpclient_tls.py` - send email using authentication and transport layer security.

IMAP clients

- `imap4client.py` - Simple IMAP4 client which displays the subjects of all messages in a particular mailbox.

Developer Guides

Sending Mail

Twisted contains many ways of sending email, but the simplest is sendmail. Intended as a near drop-in replacement of `smtplib.SMTP`'s `sendmail` method, it provides the ability to send email over SMTP/ESMTP with minimal fuss or configuration.

Knowledge of Twisted's Deferreds is required for making full use of this document.

Sending an Email over SMTP

Although insecure, some email systems still use plain SMTP for sending email. Plain SMTP has no authentication, no transport security (emails are transmitted in plain text), and should not be done over untrusted networks.

`sendmail`'s positional arguments are, in order:

- The SMTP/ESMTP server you are sending the message to
- The email address you are sending from
- A `list` of email addresses you are sending to
- The message.

The following example shows these in action.

`sendmail_smtp.py`

```
from __future__ import print_function

from twisted.mail.smtp import sendmail
from twisted.internet.task import react

def main(reactor):
    d = sendmail("myinsecuremailserver.example.com",
                 "alice@example.com",
                 ["bob@gmail.com", "charlie@gmail.com"],
                 "This is my super awesome email, sent with Twisted!")

    d.addBoth(print)
    return d

react(main)
```

Assuming that the values in it were replaced with real emails and a real SMTP server, it would send an email to the two addresses specified and print the return status.

Sending an Email over ESMTP

Extended SMTP (ESMTP) is an improved version of SMTP, and is used by most modern mail servers. Unlike SMTP, ESMTP supports authentication and transport security (emails are encrypted in transit). If you wish to send mail through services like GMail/Google Apps or Outlook.com/Office 365, you will have to use ESMTP.

Using ESMTP requires more options – usually the default port of 25 is not open, so you must find out your email provider's TLS-enabled ESMTP port. It also allows the use of authentication via a username and password.

The following example shows part of the ESMTP functionality of `sendmail`.

`sendmail_gmail.py`

```
from __future__ import print_function

from twisted.mail.smtp import sendmail
from twisted.internet.task import react

def main(reactor):
```

```
    d = sendmail("smtp.gmail.com",
                 "alice@gmail.com",
                 ["bob@gmail.com", "charlie@gmail.com"],
                 "This is my super awesome email, sent with Twisted!",
                 port=587, username="alice@gmail.com", password="*********")

    d.addBoth(print)
    return d

react(main)
```

Assuming you own the account `alice@gmail.com`, this would send an email to both `bob@gmail.com` and `charlie@gmail.com`, and print out something like the following (formatted for clarity):

```
(2, [('bob@gmail.com', 250, '2.1.5 OK hz13sm11691456pac.6 - gsmtp'),
     ('charlie@gmail.com', 250, '2.1.5 OK hz13sm11691456pac.6 - gsmtp')])
```

`sendmail` returns a 2-tuple, containing the number of emails sent successfully (note that this is from you to the server you specified, not to the recipient – emails may still be lost between that server and the recipient) and a list of statuses of the sent mail. Each status is a 3-tuple containing the address it was sent to, the SMTP status code, and the server response.

Sending Complex Emails

Sometimes you want to send more complicated emails – ones with headers, or with attachments. `sendmail` supports using Python's `email.Message`, which lets you make complex emails:

`sendmail_message.py`

```
from __future__ import print_function

from twisted.mail.smtp import sendmail
from twisted.internet.task import react

from email.mime.text import MIMEText

def main(reactor):
    me = "alice@gmail.com"
    to = ["bob@gmail.com", "charlie@gmail.com"]

    message = MIMEText("This is my super awesome email, sent with Twisted!")
    message["Subject"] = "Twisted is great!"
    message["From"] = me
    message["To"] = ", ".join(to)

    d = sendmail("smtp.gmail.com", me, to, message,
                 port=587, username=me, password="*********",
                 requireAuthentication=True,
                 requireTransportSecurity=True)

    d.addBoth(print)
    return d

react(main)
```

For more information on how to use `Message`, please see the module's Python docs.

Enforcing Transport Security

To prevent downgrade attacks, you can pass `requireTransportSecurity=True` to `sendmail`. This means that your emails will not be transmitted in plain text.

For example:

```
sendmail("smtp.gmail.com", me, to, message,
         port=587, username=me, password="*********",
         requireTransportSecurity=True)
```

Conclusion

In this document, you have seen how to:

1. Send an email over SMTP using sendmail.
2. Send an email over encrypted & authenticated ESMTP with sendmail.
3. Send a "complex" email containing a subject line using the stdlib's `email.Message` functionality.
4. Enforce transport security for emails sent using sendmail.

- *Sending Mail*: Sending mail with Twisted

Twisted Mail Tutorial: Building an SMTP Client from Scratch

Introduction

This tutorial will walk you through the creation of an extremely simple SMTP client application. By the time the tutorial is complete, you will understand how to create and start a TCP client speaking the SMTP protocol, have it connect to an appropriate mail exchange server, and transmit a message for delivery.

For the majority of this tutorial, `twistd` will be used to launch the application. Near the end we will explore other possibilities for starting a Twisted application. Until then, make sure that you have `twistd` installed and conveniently accessible for use in running each of the example `.tac` files.

SMTP Client 1

The first step is to create `smtpclient-1.tac` possible for use by `twistd`.

```
from twisted.application import service
```

The first line of the `.tac` file imports `twisted.application.service`, a module which contains many of the basic *service* classes and helper functions available in Twisted. In particular, we will be using the `Application` function to create a new *application service*. An *application service* simply acts as a central object on which to store certain kinds of deployment configuration.

```
application = service.Application("SMTP Client Tutorial")
```

The second line of the `.tac` file creates a new *application service* and binds it to the local name `application`. `twistd` requires this local name in each `.tac` file it runs. It uses various pieces of configuration on the object to determine its behavior. For example, `"SMTP Client Tutorial"` will be used as the name of the `.tap` file into which to serialize application state, should it be necessary to do so.

That does it for the first example. We now have enough of a `.tac` file to pass to `twistd`. If we run `smtpclient-1.tac` using the `twistd` command line:

```
twistd -ny smtpclient-1.tac
```

we are rewarded with the following output:

```
exarkun@boson:~/mail/tutorial/smtpclient$ twistd -ny smtpclient-1.tac
18:31 EST [-] Log opened.
18:31 EST [-] twistd 2.0.0 (/usr/bin/python2.4 2.4.1) starting up
18:31 EST [-] reactor class: twisted.internet.selectreactor.SelectReactor
18:31 EST [-] Loading smtpclient-1.tac...
18:31 EST [-] Loaded.
```

As we expected, not much is going on. We can shutdown this server by issuing `^C`:

```
18:34 EST [-] Received SIGINT, shutting down.
18:34 EST [-] Main loop terminated.
18:34 EST [-] Server Shut Down.
exarkun@boson:~/mail/tutorial/smtpclient$
```

SMTP Client 2

The first version of our SMTP client wasn't very interesting. It didn't even establish any TCP connections! The `smtpclient-2.tac` will come a little bit closer to that level of complexity. First, we need to import a few more things:

```
from twisted.application import internet
from twisted.internet import protocol
```

`twisted.application.internet` is another *application service* module. It provides services for establishing outgoing connections (as well as creating network servers, though we are not interested in those parts for the moment). `twisted.internet.protocol` provides base implementations of many of the core Twisted concepts, such as *factories* and *protocols*.

The next line of `smtpclient-2.tac` instantiates a new *client factory*.

```
smtpClientFactory = protocol.ClientFactory()
```

Client factories are responsible for constructing *protocol instances* whenever connections are established. They may be required to create just one instance, or many instances if many different connections are established, or they may never be required to create one at all, if no connection ever manages to be established.

Now that we have a client factory, we'll need to hook it up to the network somehow. The next line of `smtpclient-2.tac` does just that:

```
smtpClientService = internet.TCPClient(None, None, smtpClientFactory)
```

We'll ignore the first two arguments to `internet.TCPClient` for the moment and instead focus on the third. `TCPClient` is one of those *application service* classes. It creates TCP connections to a specified address and then uses its third argument, a *client factory*, to get a *protocol instance*. It then associates the TCP connection with the protocol instance and gets out of the way.

We can try to run `smtpclient-2.tac` the same way we ran `smtpclient-1.tac`, but the results might be a little disappointing:

```
exarkun@boson:~/mail/tutorial/smtpclient$ twistd -ny smtpclient-2.tac
18:55 EST [-] Log opened.
18:55 EST [-] twistd SVN-Trunk (/usr/bin/python2.4 2.4.1) starting up
18:55 EST [-] reactor class: twisted.internet.selectreactor.SelectReactor
18:55 EST [-] Loading smtpclient-2.tac...
18:55 EST [-] Loaded.
18:55 EST [-] Starting factory <twisted.internet.protocol.ClientFactory
              instance at 0xb791e46c>
18:55 EST [-] Traceback (most recent call last):
          File "twisted/scripts/twistd.py", line 187, in runApp
            app.runReactorWithLogging(config, oldstdout, oldstderr)
          File "twisted/application/app.py", line 128, in runReactorWithLogging
            reactor.run()
          File "twisted/internet/posixbase.py", line 200, in run
            self.mainLoop()
          File "twisted/internet/posixbase.py", line 208, in mainLoop
            self.runUntilCurrent()
        --- <exception caught here> ---
          File "twisted/internet/base.py", line 533, in runUntilCurrent
            call.func(*call.args, **call.kw)
          File "twisted/internet/tcp.py", line 489, in resolveAddress
            if abstract.isIPAddress(self.addr[0]):
          File "twisted/internet/abstract.py", line 315, in isIPAddress
            parts = string.split(addr, '.')
          File "/usr/lib/python2.4/string.py", line 292, in split
            return s.split(sep, maxsplit)
        exceptions.AttributeError: 'NoneType' object has no attribute 'split'

18:55 EST [-] Received SIGINT, shutting down.
18:55 EST [-] Main loop terminated.
18:55 EST [-] Server Shut Down.
exarkun@boson:~/mail/tutorial/smtpclient$
```

What happened? Those first two arguments to `TCPClient` turned out to be important after all. We'll get to them in the next example.

SMTP Client 3

Version three of our SMTP client only changes one thing. The line from version two:

```
smtpClientService = internet.TCPClient(None, None, smtpClientFactory)
```

has its first two arguments changed from `None` to something with a bit more meaning:

```
smtpClientService = internet.TCPClient('localhost', 25, smtpClientFactory)
```

This directs the client to connect to *localhost* on port *25* . This isn't the address we want ultimately, but it's a good place-holder for the time being. We can run `smtpclient-3.tac` and see what this change gets us:

```
exarkun@boson:~/mail/tutorial/smtpclient$ twistd -ny smtpclient-3.tac
19:10 EST [-] Log opened.
19:10 EST [-] twistd SVN-Trunk (/usr/bin/python2.4 2.4.1) starting up
19:10 EST [-] reactor class: twisted.internet.selectreactor.SelectReactor
19:10 EST [-] Loading smtpclient-3.tac...
19:10 EST [-] Loaded.
19:10 EST [-] Starting factory <twisted.internet.protocol.ClientFactory
              instance at 0xb791e48c>
```

```
19:10 EST [-] Enabling Multithreading.
19:10 EST [Uninitialized] Traceback (most recent call last):
          File "twisted/python/log.py", line 56, in callWithLogger
            return callWithContext({"system": lp}, func, *args, **kw)
          File "twisted/python/log.py", line 41, in callWithContext
            return context.call({ILogContext: newCtx}, func, *args, **kw)
          File "twisted/python/context.py", line 52, in callWithContext
            return self.currentContext().callWithContext(ctx, func, *args, **kw)
          File "twisted/python/context.py", line 31, in callWithContext
            return func(*args,**kw)
        --- <exception caught here> ---
          File "twisted/internet/selectreactor.py", line 139, in _doReadOrWrite
            why = getattr(selectable, method)()
          File "twisted/internet/tcp.py", line 543, in doConnect
            self._connectDone()
          File "twisted/internet/tcp.py", line 546, in _connectDone
            self.protocol = self.connector.buildProtocol(self.getPeer())
          File "twisted/internet/base.py", line 641, in buildProtocol
            return self.factory.buildProtocol(addr)
          File "twisted/internet/protocol.py", line 99, in buildProtocol
            p = self.protocol()
        exceptions.TypeError: 'NoneType' object is not callable

19:10 EST [Uninitialized] Stopping factory
          <twisted.internet.protocol.ClientFactory instance at
          0xb791e48c>
19:10 EST [-] Received SIGINT, shutting down.
19:10 EST [-] Main loop terminated.
19:10 EST [-] Server Shut Down.
exarkun@boson:~/mail/tutorial/smtpclient$
```

A meager amount of progress, but the service still raises an exception. This time, it's because we haven't specified a *protocol class* for the factory to use. We'll do that in the next example.

SMTP Client 4

In the previous example, we ran into a problem because we hadn't set up our *client factory's protocol* attribute correctly (or at all). `ClientFactory.buildProtocol` is the method responsible for creating a *protocol instance* . The default implementation calls the factory's `protocol` attribute, adds itself as an attribute named `factory` to the resulting instance, and returns it. In `smtpclient-4.tac` , we'll correct the oversight that caused the traceback in smtpclient-3.tac:

```
smtpClientFactory.protocol = protocol.Protocol
```

Running this version of the client, we can see the output is once again traceback free:

```
exarkun@boson:~/doc/mail/tutorial/smtpclient$ twistd -ny smtpclient-4.tac
19:29 EST [-] Log opened.
19:29 EST [-] twistd SVN-Trunk (/usr/bin/python2.4 2.4.1) starting up
19:29 EST [-] reactor class: twisted.internet.selectreactor.SelectReactor
19:29 EST [-] Loading smtpclient-4.tac...
19:29 EST [-] Loaded.
19:29 EST [-] Starting factory <twisted.internet.protocol.ClientFactory
              instance at 0xb791e4ac>
19:29 EST [-] Enabling Multithreading.
19:29 EST [-] Received SIGINT, shutting down.
```

```
19:29 EST [Protocol,client] Stopping factory
          <twisted.internet.protocol.ClientFactory instance at
          0xb791e4ac>
19:29 EST [-] Main loop terminated.
19:29 EST [-] Server Shut Down.
exarkun@boson:~/doc/mail/tutorial/smtpclient$
```

But what does this mean? `twisted.internet.protocol.Protocol` is the base *protocol* implementation. For those familiar with the classic UNIX network services, it is equivalent to the *discard* service. It never produces any output and it discards all its input. Not terribly useful, and certainly nothing like an SMTP client. Let's see how we can improve this in the next example.

SMTP Client 5

In `smtpclient-5.tac` , we will begin to use Twisted's SMTP protocol implementation for the first time. We'll make the obvious change, simply swapping out `twisted.internet.protocol.Protocol` in favor of `twisted.mail.smtp.ESMTPClient` . Don't worry about the *E* in *ESMTP* . It indicates we're actually using a newer version of the SMTP protocol. There is an `SMTPClient` in Twisted, but there's essentially no reason to ever use it.

smtpclient-5.tac adds a new import:

```
from twisted.mail import smtp
```

All of the mail related code in Twisted exists beneath the `twisted.mail` package. More specifically, everything having to do with the SMTP protocol implementation is defined in the `twisted.mail.smtp` module.

Next we remove a line we added in smtpclient-4.tac:

```
smtpClientFactory.protocol = protocol.Protocol
```

And add a similar one in its place:

```
smtpClientFactory.protocol = smtp.ESMTPClient
```

Our client factory is now using a protocol implementation which behaves as an SMTP client. What happens when we try to run this version?

```
exarkun@boson:~/doc/mail/tutorial/smtpclient$ twistd -ny smtpclient-5.tac
19:42 EST [-] Log opened.
19:42 EST [-] twistd SVN-Trunk (/usr/bin/python2.4 2.4.1) starting up
19:42 EST [-] reactor class: twisted.internet.selectreactor.SelectReactor
19:42 EST [-] Loading smtpclient-5.tac...
19:42 EST [-] Loaded.
19:42 EST [-] Starting factory <twisted.internet.protocol.ClientFactory
              instance at 0xb791e54c>
19:42 EST [-] Enabling Multithreading.
19:42 EST [Uninitialized] Traceback (most recent call last):
          File "twisted/python/log.py", line 56, in callWithLogger
            return callWithContext({"system": lp}, func, *args, **kw)
          File "twisted/python/log.py", line 41, in callWithContext
            return context.call({ILogContext: newCtx}, func, *args, **kw)
          File "twisted/python/context.py", line 52, in callWithContext
            return self.currentContext().callWithContext(ctx, func, *args, **kw)
          File "twisted/python/context.py", line 31, in callWithContext
            return func(*args,**kw)
        --- <exception caught here> ---
```

```
          File "twisted/internet/selectreactor.py", line 139, in _doReadOrWrite
            why = getattr(selectable, method)()
          File "twisted/internet/tcp.py", line 543, in doConnect
            self._connectDone()
          File "twisted/internet/tcp.py", line 546, in _connectDone
            self.protocol = self.connector.buildProtocol(self.getPeer())
          File "twisted/internet/base.py", line 641, in buildProtocol
            return self.factory.buildProtocol(addr)
          File "twisted/internet/protocol.py", line 99, in buildProtocol
            p = self.protocol()
        exceptions.TypeError: __init__() takes at least 2 arguments (1 given)

19:42 EST [Uninitialized] Stopping factory
          <twisted.internet.protocol.ClientFactory instance at
          0xb791e54c>
19:43 EST [-] Received SIGINT, shutting down.
19:43 EST [-] Main loop terminated.
19:43 EST [-] Server Shut Down.
exarkun@boson:~/doc/mail/tutorial/smtpclient$
```

Oops, back to getting a traceback. This time, the default implementation of `buildProtocol` seems no longer to be sufficient. It instantiates the protocol with no arguments, but `ESMTPClient` wants at least one argument. In the next version of the client, we'll override `buildProtocol` to fix this problem.

SMTP Client 6

`smtpclient-6.tac` introduces a `twisted.internet.protocol.ClientFactory` subclass with an overridden `buildProtocol` method to overcome the problem encountered in the previous example.

```
class SMTPClientFactory(protocol.ClientFactory):
    protocol = smtp.ESMTPClient

    def buildProtocol(self, addr):
        return self.protocol(secret=None, identity='example.com')
```

The overridden method does almost the same thing as the base implementation: the only change is that it passes values for two arguments to `twisted.mail.smtp.ESMTPClient` 's initializer. The `secret` argument is used for SMTP authentication (which we will not attempt yet). The `identity` argument is used as a to identify ourselves Another minor change to note is that the `protocol` attribute is now defined in the class definition, rather than tacked onto an instance after one is created. This means it is a class attribute, rather than an instance attribute, now, which makes no difference as far as this example is concerned. There are circumstances in which the difference is important: be sure you understand the implications of each approach when creating your own factories.

One other change is required: instead of instantiating `twisted.internet.protocol.ClientFactory` , we will now instantiate `SMTPClientFactory` :

```
smtpClientFactory = SMTPClientFactory()
```

Running this version of the code, we observe that the code **still** isn't quite traceback-free.

```
exarkun@boson:~/doc/mail/tutorial/smtpclient$ twistd -ny smtpclient-6.tac
21:17 EST [-] Log opened.
21:17 EST [-] twistd SVN-Trunk (/usr/bin/python2.4 2.4.1) starting up
21:17 EST [-] reactor class: twisted.internet.selectreactor.SelectReactor
21:17 EST [-] Loading smtpclient-6.tac...
21:17 EST [-] Loaded.
```

```
21:17 EST [-] Starting factory <__builtin__.SMTPClientFactory instance
              at 0xb77fd68c>
21:17 EST [-] Enabling Multithreading.
21:17 EST [ESMTPClient,client] Traceback (most recent call last):
          File "twisted/python/log.py", line 56, in callWithLogger
            return callWithContext({"system": lp}, func, *args, **kw)
          File "twisted/python/log.py", line 41, in callWithContext
            return context.call({ILogContext: newCtx}, func, *args, **kw)
          File "twisted/python/context.py", line 52, in callWithContext
            return self.currentContext().callWithContext(ctx, func, *args, **kw)
          File "twisted/python/context.py", line 31, in callWithContext
            return func(*args,**kw)
        --- <exception caught here> ---
          File "twisted/internet/selectreactor.py", line 139, in _doReadOrWrite
            why = getattr(selectable, method)()
          File "twisted/internet/tcp.py", line 351, in doRead
            return self.protocol.dataReceived(data)
          File "twisted/protocols/basic.py", line 221, in dataReceived
            why = self.lineReceived(line)
          File "twisted/mail/smtp.py", line 1039, in lineReceived
            why = self._okresponse(self.code,'\n'.join(self.resp))
          File "twisted/mail/smtp.py", line 1281, in esmtpState_serverConfig
            self.tryTLS(code, resp, items)
          File "twisted/mail/smtp.py", line 1294, in tryTLS
            self.authenticate(code, resp, items)
          File "twisted/mail/smtp.py", line 1343, in authenticate
            self.smtpState_from(code, resp)
          File "twisted/mail/smtp.py", line 1062, in smtpState_from
            self._from = self.getMailFrom()
          File "twisted/mail/smtp.py", line 1137, in getMailFrom
            raise NotImplementedError
        exceptions.NotImplementedError:

21:17 EST [ESMTPClient,client] Stopping factory
          <__builtin__.SMTPClientFactory instance at 0xb77fd68c>
21:17 EST [-] Received SIGINT, shutting down.
21:17 EST [-] Main loop terminated.
21:17 EST [-] Server Shut Down.
exarkun@boson:~/doc/mail/tutorial/smtpclient$
```

What we have accomplished with this iteration of the example is to navigate far enough into an SMTP transaction that Twisted is now interested in calling back to application-level code to determine what its next step should be. In the next example, we'll see how to provide that information to it.

SMTP Client 7

SMTP Client 7 is the first version of our SMTP client which actually includes message data to transmit. For simplicity's sake, the message is defined as part of a new class. In a useful program which sent email, message data might be pulled in from the filesystem, a database, or be generated based on user-input. `smtpclient-7.tac`, however, defines a new class, `SMTPTutorialClient`, with three class attributes (`mailFrom`, `mailTo`, and `mailData`):

```
class SMTPTutorialClient(smtp.ESMTPClient):
    mailFrom = "tutorial_sender@example.com"
    mailTo = "tutorial_recipient@example.net"
    mailData = '''\
```

```
Date: Fri, 6 Feb 2004 10:14:39 -0800
From: Tutorial Guy <tutorial_sender@example.com>
To: Tutorial Gal <tutorial_recipient@example.net>
Subject: Tutorate!

Hello, how are you, goodbye.
'''
```

This statically defined data is accessed later in the class definition by three of the methods which are part of the *SMTPClient callback API* . Twisted expects each of the three methods below to be defined and to return an object with a particular meaning. First, `getMailFrom` :

```
def getMailFrom(self):
    result = self.mailFrom
    self.mailFrom = None
    return result
```

This method is called to determine the *reverse-path* , otherwise known as the *envelope from* , of the message. This value will be used when sending the `MAIL FROM` SMTP command. The method must return a string which conforms to the RFC 2821 definition of a *reverse-path* . In simpler terms, it should be a string like `"alice@example.com"` . Only one *envelope from* is allowed by the SMTP protocol, so it cannot be a list of strings or a comma separated list of addresses. Our implementation of `getMailFrom` does a little bit more than just return a string; we'll get back to this in a little bit.

The next method is `getMailTo` :

```
def getMailTo(self):
    return [self.mailTo]
```

`getMailTo` is similar to `getMailFrom` . It returns one or more RFC 2821 addresses (this time a *forward-path* , or *envelope to*). Since SMTP allows multiple recipients, `getMailTo` returns a list of these addresses. The list must contain at least one address, and even if there is exactly one recipient, it must still be in a list.

The final callback we will define to provide information to Twisted is `getMailData` :

```
def getMailData(self):
    return StringIO.StringIO(self.mailData)
```

This one is quite simple as well: it returns a file or a file-like object which contains the message contents. In our case, we return a `StringIO` since we already have a string containing our message. If the contents of the file returned by `getMailData` span multiple lines (as email messages often do), the lines should be `\n` delimited (as they would be when opening a text file in the `"rt"` mode): necessary newline translation will be performed by `SMTPClient` automatically.

There is one more new callback method defined in smtpclient-7.tac. This one isn't for providing information about the messages to Twisted, but for Twisted to provide information about the success or failure of the message transmission to the application:

```
def sentMail(self, code, resp, numOk, addresses, log):
    print('Sent', numOk, 'messages')
```

Each of the arguments to `sentMail` provides some information about the success or failure of the message transmission transaction. `code` is the response code from the ultimate command. For successful transactions, it will be 250. For transient failures (those which should be retried), it will be between 400 and 499, inclusive. For permanent failures (this which will never work, no matter how many times you retry them), it will be between 500 and 599.

SMTP Client 8

Thus far we have succeeded in creating a Twisted client application which starts up, connects to a (possibly) remote host, transmits some data, and disconnects. Notably missing, however, is application shutdown. Hitting ^C is fine during development, but it's not exactly a long-term solution. Fortunately, programmatic shutdown is extremely simple. `smtpclient-8.tac` extends `sentMail` with these two lines:

```
from twisted.internet import reactor
reactor.stop()
```

The `stop` method of the reactor causes the main event loop to exit, allowing a Twisted server to shut down. With this version of the example, we see that the program actually terminates after sending the message, without user-intervention:

```
exarkun@boson:~/doc/mail/tutorial/smtpclient$ twistd -ny smtpclient-8.tac
19:52 EST [-] Log opened.
19:52 EST [-] twistd SVN-Trunk (/usr/bin/python2.4 2.4.1) starting up
19:52 EST [-] reactor class: twisted.internet.selectreactor.SelectReactor
19:52 EST [-] Loading smtpclient-8.tac...
19:52 EST [-] Loaded.
19:52 EST [-] Starting factory <__builtin__.SMTPClientFactory instance
              at 0xb791beec>
19:52 EST [-] Enabling Multithreading.
19:52 EST [SMTPTutorialClient,client] Sent 1 messages
19:52 EST [SMTPTutorialClient,client] Stopping factory
          <__builtin__.SMTPClientFactory instance at 0xb791beec>
19:52 EST [-] Main loop terminated.
19:52 EST [-] Server Shut Down.
exarkun@boson:~/doc/mail/tutorial/smtpclient$
```

SMTP Client 9

One task remains to be completed in this tutorial SMTP client: instead of always sending mail through a well-known host, we will look up the mail exchange server for the recipient address and try to deliver the message to that host.

In `smtpclient-9.tac` , we'll take the first step towards this feature by defining a function which returns the mail exchange host for a particular domain:

```
def getMailExchange(host):
    return 'localhost'
```

Obviously this doesn't return the correct mail exchange host yet (in fact, it returns the exact same host we have been using all along), but pulling out the logic for determining which host to connect to into a function like this is the first step towards our ultimate goal. Now that we have `getMailExchange` , we'll call it when constructing our `TCPClient` service:

```
smtpClientService = internet.TCPClient(
    getMailExchange('example.net'), 25, smtpClientFactory)
```

We'll expand on the definition of `getMailExchange` in the next example.

SMTP Client 10

In the previous example we defined `getMailExchange` to return a string representing the mail exchange host for a particular domain. While this was a step in the right direction, it turns out not to be a very big one. Determining

the mail exchange host for a particular domain is going to involve network traffic (specifically, some DNS requests). These might take an arbitrarily large amount of time, so we need to introduce a `Deferred` to represent the result of `getMailExchange` . `smtpclient-10.tac` redefines it thusly:

```
def getMailExchange(host):
    return defer.succeed('localhost')
```

`defer.succeed` is a function which creates a new `Deferred` which already has a result, in this case `'localhost'` . Now we need to adjust our `TCPClient` -constructing code to expect and properly handle this `Deferred` :

```
def cbMailExchange(exchange):
    smtpClientFactory = SMTPClientFactory()

    smtpClientService = internet.TCPClient(exchange, 25, smtpClientFactory)
    smtpClientService.setServiceParent(application)

getMailExchange('example.net').addCallback(cbMailExchange)
```

An in-depth exploration of `Deferred` s is beyond the scope of this document. For such a look, see the Deferred Reference `TCPClient` until the `Deferred` returned by `getMailExchange` fires. Once it does, we proceed normally through the creation of our `SMTPClientFactory` and `TCPClient` , as well as set the `TCPClient` 's service parent, just as we did in the previous examples.

SMTP Client 11

At last we're ready to perform the mail exchange lookup. We do this by calling on an object provided specifically for this task, `twisted.mail.relaymanager.MXCalculator` :

```
def getMailExchange(host):
    def cbMX(mxRecord):
        return str(mxRecord.name)
    return relaymanager.MXCalculator().getMX(host).addCallback(cbMX)
```

Because `getMX` returns a `Record_MX` object rather than a string, we do a little bit of post-processing to get the results we want. We have already converted the rest of the tutorial application to expect a `Deferred` from `getMailExchange` , so no further changes are required. `smtpclient-11.tac` completes this tutorial by being able to both look up the mail exchange host for the recipient domain, connect to it, complete an SMTP transaction, report its results, and finally shut down the reactor.

- *Examples*: short code examples using Twisted Mail
- *Developer Guides*: documentation on using Twisted Mail
- *Twisted Mail Tutorial*: Building an SMTP Client from Scratch

CHAPTER 5

Twisted Names

Developer Guides

A Guided Tour of twisted.names.client

Twisted Names provides a layered selection of client APIs.

In this section you will learn:

- about the high level client API,
- about how you can use the client API interactively from the Python shell (useful for DNS debugging and diagnostics),
- about the IResolverSimple and the IResolver interfaces,
- about various implementations of those interfaces and when to use them,
- how to customise how the reactor carries out hostname resolution,
- and finally, you will also be introduced to some of the low level APIs.

Using the Global Resolver

The easiest way to issue DNS queries from Twisted is to use the module level functions in names.client.

Here's an example showing some DNS queries generated in an interactive `twisted.conch` shell.

Note: The `twisted.conch` shell starts a `reactor` so that asynchronous operations can be run interactively and it prints the current result of `deferred`s which have fired.

You'll notice that the `deferred`s returned in the following examples do not immediately have a result – they are waiting for a response from the DNS server.

So we type _ (the default variable) a little later, to display the value of the `deferred` after an answer has been received and the `deferred` has fired.

```
$ python -m twisted.conch.stdio
```

```
>>> from twisted.names import client
>>> client.getHostByName('www.example.com')
<Deferred at 0xf5c5a8 waiting on Deferred at 0xf5cb90>
>>> _
<Deferred at 0xf5c5a8 current result: '2606:2800:220:6d:26bf:1447:1097:aa7'>

>>> client.lookupMailExchange('twistedmatrix.com')
<Deferred at 0xf5cd40 waiting on Deferred at 0xf5cea8>
>>> _
<Deferred at 0xf5cd40 current result: ([<RR name=twistedmatrix.com type=MX class=IN␣
↪ttl=1s auth=False>], [], [])>
```

All the IResolverSimple and IResolver methods are asynchronous and therefore return `deferreds`.

getHostByName (part of IResolverSimple) returns an IP address whereas lookupMailExchange returns three lists of DNS records. These three lists contain answer records, authority records, and additional records.

Note:

- getHostByName may return an IPv6 address; unlike its stdlib equivalent (`socket.gethostbyname()`)
- IResolver contains separate functions for looking up each of the common DNS record types.
- IResolver includes a lower level `query` function for issuing arbitrary queries.
- The names.client module `directlyProvides` both the IResolverSimple and the IResolver interfaces.
- createResolver constructs a global resolver which performs queries against the same DNS sources and servers used by the underlying operating system.

  That is, it will use the DNS server IP addresses found in a local `resolv.conf` file (if the operating system provides such a file) and it will use an OS specific `hosts` file path.

A simple example

In this section you will learn how the IResolver interface can be used to write a utility for performing a reverse DNS lookup for an IPv4 address. dig can do this too, so lets start by examining its output:

```
$ dig -x 127.0.0.1
...
;; QUESTION SECTION:
;1.0.0.127.in-addr.arpa.             IN      PTR

;; ANSWER SECTION:
1.0.0.127.in-addr.arpa.      86400   IN      PTR     localhost.
...
```

As you can see, `dig` has performed a DNS query with the following attributes:

- Name: `1.0.0.127.in-addr.arpa.`
- Class: `IN`

- Type: `PTR`

The *name* is a **reverse domain name** and is derived by reversing an IPv4 address and prepending it to the special *in-addr.arpa* parent domain name. So, lets write a function to create a reverse domain name from an IP address.

```
def reverseNameFromIPAddress(address):
    return '.'.join(reversed(address.split('.'))) + '.in-addr.arpa'
```

We can test the output from a python shell:

```
>>> reverseNameFromIPAddress('192.0.2.100')
'100.2.0.192.in-addr.arpa'
```

We're going to use twisted.names.client.lookupPointer to perform the actual DNS lookup. So lets examine the output of `lookupPointer` so that we can design a function to format and print its results in a style similar to `dig`.

Note: `lookupPointer` is an asynchronous function, so we'll use an interactive `twisted.conch` shell here.

```
$ python -m twisted.conch.stdio
```

```
>>> from twisted.names import client
>>> from reverse_lookup import reverseNameFromIPAddress
>>> d = client.lookupPointer(name=reverseNameFromIPAddress('127.0.0.1'))
>>> d
<Deferred at 0x286b170 current result: ([<RR name=1.0.0.127.in-addr.arpa type=PTR␣
↪class=IN ttl=86400s auth=False>], [], [])>
>>> d.result
([<RR name=1.0.0.127.in-addr.arpa type=PTR class=IN ttl=86400s auth=False>], [], [])
```

The deferred result of `lookupPointer` is a tuple containing three lists of records; **answers**, **authority**, and **additional**. The actual record is a Record_PTR instance which can be reached via the RRHeader `.payload` attribute.

```
>>> recordHeader = d.result[0][0]
>>> recordHeader.payload
<PTR name=localhost ttl=86400>
```

So, now we've found the information we need, lets create a function that extracts the first *answer* and prints the domain name and the record payload.

```
def printResult(result):
    answers, authority, additional = result
    if answers:
        a = answers[0]
        print('{} IN {}'.format(a.name.name, a.payload))
```

And lets test the output:

```
>>> from twisted.names import dns
>>> printResult(([dns.RRHeader(name='1.0.0.127.in-addr.arpa', type=dns.PTR,␣
↪payload=dns.Record_PTR('localhost'))], [], []))
1.0.0.127.in-addr.arpa IN <PTR name=localhost ttl=None>
```

Fine! Now we can assemble the pieces in a `main` function, which we'll call using twisted.internet.task.react. Here's the complete script.

`listings/names/reverse_lookup.py`

```
from __future__ import print_function

import sys

from twisted.internet import task
from twisted.names import client

def reverseNameFromIPAddress(address):
    return '.'.join(reversed(address.split('.'))) + '.in-addr.arpa'

def printResult(result):
    answers, authority, additional = result
    if answers:
        a = answers[0]
        print('{} IN {}'.format(a.name.name, a.payload))

def main(reactor, address):
    d = client.lookupPointer(name=reverseNameFromIPAddress(address=address))
    d.addCallback(printResult)
    return d

task.react(main, sys.argv[1:])
```

The output looks like this:

```
$ python reverse_lookup.py 127.0.0.1
1.0.0.127.in-addr.arpa IN <PTR name=localhost ttl=86400>
```

Note:

- You can read more about reverse domain names in **RFC 1034#section-5.2.1**.
- We've ignored IPv6 addresses in this example, but you can read more about reverse IPv6 domain names in **RFC 3596#section-2.5** and the example could easily be extended to support these.
- You might also consider using netaddr, which can generate reverse domain names and which also includes sophisticated IP network and IP address handling.
- This script only prints the first answer, but sometimes you'll get multiple answers due to CNAME indirection, for example in the case of classless reverse zones.
- All lookups and responses are handled asynchronously, so the script could be extended to perform thousands of reverse DNS lookups in parallel.

Next you should study `../examples/multi_reverse_lookup.py` which extends this example to perform both IPv4 and IPv6 addresses and which can perform multiple reverse DNS lookups in parallel.

Creating a New Resolver

Now suppose we want to create a DNS client which sends its queries to a specific server (or servers).

In this case, we use client.Resolver directly and pass it a list of preferred server IP addresses and ports.

For example, suppose we want to lookup names using the free Google DNS servers:

```
$ python -m twisted.conch.stdio
```

```
>>> from twisted.names import client
>>> resolver = client.createResolver(servers=[('8.8.8.8', 53), ('8.8.4.4', 53)])
>>> resolver.getHostByName('example.com')
<Deferred at 0x9dcfbac current result: '93.184.216.119'>
```

Here we are using the Google DNS server IP addresses and the standard DNS port (53).

Installing a Resolver in the Reactor

You can also install a custom resolver into the reactor using the IReactorPluggable interface.

The reactor uses its installed resolver whenever it needs to resolve hostnames; for example, when you supply a hostname to connectTCP.

Here's a short example that shows how to install an alternative resolver for the global reactor:

```
from twisted.internet import reactor
from twisted.names import client
reactor.installResolver(client.createResolver(servers=[('8.8.8.8', 53), ('8.8.4.4',␣
↪53)]))
```

After this, all hostname lookups requested by the reactor will be sent to the Google DNS servers; instead of to the local operating system.

Note:

- By default the reactor uses the POSIX `gethostbyname` function provided by the operating system,
- but `gethostbyname` is a blocking function, so it has to be called in a thread pool.
- Check out ThreadedResolver if you're interested in learning more about how the default threaded resolver works.

Lower Level APIs

Here's an example of how to use the DNSDatagramProtocol directly.

```
from twisted.internet import task
from twisted.names import dns

def main(reactor):
    proto = dns.DNSDatagramProtocol(controller=None)
    reactor.listenUDP(0, proto)

    d = proto.query(('8.8.8.8', 53), [dns.Query('www.example.com', dns.AAAA)])
    d.addCallback(printResult)
    return d

def printResult(res):
    print('ANSWERS: ', [a.payload for a in res.answers])

task.react(main)
```

The disadvantage of working at this low level is that you will need to handle query failures yourself, by manually re-issuing queries or by issuing followup TCP queries using the stream based dns.DNSProtocol.

These things are handled automatically by the higher level APIs in client.

Also notice that in this case, the deferred result of dns.DNSDatagramProtocol.query is a dns.Message object, rather than a list of DNS records.

Further Reading

Check out the *Twisted Names Examples* which demonstrate how the client APIs can be used to create useful DNS diagnostic tools.

Creating and working with a names (DNS) server

A Names server can be perform three basic operations:

- act as a recursive server, forwarding queries to other servers
- perform local caching of recursively discovered records
- act as the authoritative server for a domain

Creating a non-authoritative server

The first two of these are easy, and you can create a server that performs them with the command `twistd -n dns --recursive --cache`. You may wish to run this as root since it will try to bind to UDP port 53. Try performing a lookup with it, `dig twistedmatrix.com @127.0.0.1`.

Creating an authoritative server

To act as the authority for a domain, two things are necessary: the address of the machine on which the domain name server will run must be registered as a nameserver for the domain; and the domain name server must be configured to act as the authority. The first requirement is beyond the scope of this howto and will not be covered.

To configure Names to act as the authority for `example-domain.com`, we first create a zone file for this domain.

`example-domain.com`

```
zone = [
    SOA(
        # For whom we are the authority
        'example-domain.com',

        # This nameserver's name
        mname = "ns1.example-domain.com",

        # Mailbox of individual who handles this
        rname = "root.example-domain.com",

        # Unique serial identifying this SOA data
        serial = 2003010601,

        # Time interval before zone should be refreshed
        refresh = "1H",

        # Interval before failed refresh should be retried
        retry = "1H",
```

```
        # Upper limit on time interval before expiry
        expire = "1H",

        # Minimum TTL
        minimum = "1H"
    ),

    A('example-domain.com', '127.0.0.1'),
    NS('example-domain.com', 'ns1.example-domain.com'),

    CNAME('www.example-domain.com', 'example-domain.com'),
    CNAME('ftp.example-domain.com', 'example-domain.com'),

    MX('example-domain.com', 0, 'mail.example-domain.com'),
    A('mail.example-domain.com', '123.0.16.43')
]
```

Next, run the command `twistd -n dns --pyzone example-domain.com`. Now try querying the domain locally (again, with dig): `dig -t any example-domain.com @127.0.0.1`.

Names can also read a traditional, BIND-syntax zone file. Specify these with the `--bindzone` parameter. The $GENERATE and $INCLUDE directives are not yet supported.

Creating a custom server

The builtin DNS server plugin is useful, but the beauty of Twisted Names is that you can build your own custom servers and clients using the names components.

- In this section you will learn about the components required to build a simple DNS server.
- You will then learn how to create a custom DNS server which calculates responses dynamically.

A simple forwarding DNS server

Lets start by creating a simple forwarding DNS server, which forwards all requests to an upstream server (or servers).

`simple_server.py`

```
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

"""
An example of a simple non-authoritative DNS server.
"""

from twisted.internet import reactor
from twisted.names import client, dns, server


def main():
    """
    Run the server.
    """
    factory = server.DNSServerFactory(
        clients=[client.Resolver(resolv='/etc/resolv.conf')]
    )
```

```
    protocol = dns.DNSDatagramProtocol(controller=factory)

    reactor.listenUDP(10053, protocol)
    reactor.listenTCP(10053, factory)

    reactor.run()


if __name__ == '__main__':
    raise SystemExit(main())
```

In this example we are passing a client.Resolver instance to the DNSServerFactory and we are configuring that client to use the upstream DNS servers which are specified in a local `resolv.conf` file.

Also note that we start the server listening on both UDP and TCP ports. This is a standard requirement for DNS servers.

You can test the server using `dig`. For example:

```
$ dig -p 10053 @127.0.0.1 example.com SOA +short
sns.dns.icann.org. noc.dns.icann.org. 2013102791 7200 3600 1209600 3600
```

A server which computes responses dynamically

Now suppose we want to create a bespoke DNS server which responds to certain hostname queries by dynamically calculating the resulting IP address, while passing all other queries to another DNS server. Queries for hostnames matching the pattern **workstation{0-9}+** will result in an IP address where the last octet matches the workstation number.

We'll write a custom resolver which we insert before the standard client resolver. The custom resolver will be queried first.

Here's the code:

`override_server.py`

```
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

"""
An example demonstrating how to create a custom DNS server.

The server will calculate the responses to A queries where the name begins with
the word "workstation".

Other queries will be handled by a fallback resolver.

eg
    python doc/names/howto/listings/names/override_server.py

    $ dig -p 10053 @localhost workstation1.example.com A +short
    172.0.2.1
"""

from twisted.internet import reactor, defer
from twisted.names import client, dns, error, server
```

```
class DynamicResolver(object):
    """
    A resolver which calculates the answers to certain queries based on the
    query type and name.
    """
    _pattern = 'workstation'
    _network = '172.0.2'

    def _dynamicResponseRequired(self, query):
        """
        Check the query to determine if a dynamic response is required.
        """
        if query.type == dns.A:
            labels = query.name.name.split('.')
            if labels[0].startswith(self._pattern):
                return True

        return False


    def _doDynamicResponse(self, query):
        """
        Calculate the response to a query.
        """
        name = query.name.name
        labels = name.split('.')
        parts = labels[0].split(self._pattern)
        lastOctet = int(parts[1])
        answer = dns.RRHeader(
            name=name,
            payload=dns.Record_A(address=b'%s.%s' % (self._network, lastOctet)))
        answers = [answer]
        authority = []
        additional = []
        return answers, authority, additional


    def query(self, query, timeout=None):
        """
        Check if the query should be answered dynamically, otherwise dispatch to
        the fallback resolver.
        """
        if self._dynamicResponseRequired(query):
            return defer.succeed(self._doDynamicResponse(query))
        else:
            return defer.fail(error.DomainError())



def main():
    """
    Run the server.
    """
    factory = server.DNSServerFactory(
        clients=[DynamicResolver(), client.Resolver(resolv='/etc/resolv.conf')]
    )
```

```
    protocol = dns.DNSDatagramProtocol(controller=factory)

    reactor.listenUDP(10053, protocol)
    reactor.listenTCP(10053, factory)

    reactor.run()



if __name__ == '__main__':
    raise SystemExit(main())
```

Notice that `DynamicResolver.query` returns a Deferred. On success, it returns three lists of DNS records (answers, authority, additional), which will be encoded by dns.Message and returned to the client. On failure, it returns a DomainError, which is a signal that the query should be dispatched to the next client resolver in the list.

Note: The fallback behaviour is actually handled by ResolverChain.

ResolverChain is a proxy for other resolvers. It takes a list of IResolver providers and queries each one in turn until it receives an answer, or until the list is exhausted.

Each IResolver in the chain may return a deferred DomainError, which is a signal that ResolverChain should query the next chained resolver.

The DNSServerFactory constructor takes a list of authoritative resolvers, caches and client resolvers and ensures that they are added to the ResolverChain in the correct order.

Let's use `dig` to see how this server responds to requests that match the pattern we specified:

```
$ dig -p 10053 @127.0.0.1 workstation1.example.com A +short
172.0.2.1

$ dig -p 10053 @127.0.0.1 workstation100.example.com A +short
172.0.2.100
```

And if we issue a request that doesn't match the pattern:

```
$ dig -p 10053 @localhost www.example.com A +short
93.184.216.119
```

Further Reading

For simplicity, the examples above use the `reactor.listenXXX` APIs. But your application will be more flexible if you use the *Twisted Application APIs*, along with the *Twisted plugin system* and `twistd`. Read the source code of names.tap to see how the `twistd names` plugin works.

- *A guided tour of twisted.names.client*
- *Using the twistd plugin*
- *Create a custom DNS server*

Examples

DNS (Twisted Names)

- `testdns.py` - Prints the results of an Address record lookup, Mail-Exchanger record lookup, and Nameserver record lookup for the given domain name.
- `dns-service.py` - Searches for SRV records in DNS.
- `gethostbyname.py` - Returns the IP address for a given hostname.

Twisted Names is a library of DNS components for building DNS servers and clients.

It includes a client resolver API, with which you can generate queries for all the standard record types. The client API also includes a replacement for the blocking `gethostbyname()` function provided by the Python stdlib socket module.

Twisted Names provides a `twistd` DNS server plugin which can:

- Act as a master authoritative server which can read most BIND-syntax zone files as well as a simple Python-based configuration format.
- Act as a secondary authoritative DNS server, which retrieves its records from a master server by zone transfer.
- Act as a caching / forwarding nameserver which forwards requests to one or more upstream recursive nameservers and caches the results.
- Or any combination of these.

The following developer guides, example scripts and API documentation will demonstrate how to use these components and provide you with all the information you need to build your own custom DNS client or server using Twisted Names.

- *Developer guides*: documentation on using Twisted Names to develop your own applications
- *Examples*: short code examples using Twisted Names
- API documentation: Detailed API documentation for all the Twisted Names components

CHAPTER 6

Twisted Pair

Developer Guides

Twisted Pair: Tunnels And Network Taps

On Linux, Twisted Pair supports the special *tun* and *tap* network interface types. This functionality allows you to interact with raw sockets (for example, to send or receive ICMP or ARP traffic). It also allows the creation of simulated networks. This document will not cover the details of these platform-provided features, but it will explain how to use the Twisted Pair APIs which interact with them. Before reading this document, you may want to familiarize yourself with Linux tuntap if you have not already done so (good online resources, are a little scarce, but you may find the linux tuntap tutorial google results helpful).

Tuntap Ports

The twisted.pair.tuntap.TuntapPort class is the entry point into the tun/tap functionality. This class is initialized with an application-supplied protocol object and associates that object with a tun or tap device on the system. If the protocol provides twisted.pair.ethernet.IEthernetProtocol then it is associated with a tap device. Otherwise the protocol must provide twisted.pair.raw.IRawPacketProtocol and it will be associated with a tun device.

```
from zope.interface import implementer
from twisted.pair.tuntap import TuntapPort
from twisted.pair.ethernet import EthernetProtocol
from twisted.pair.rawudp import RawUDPProtocol
from twisted.internet import reactor

# Note that you must run this example as a user with permission to open this
# device.  That means run it as root or pre-configure tap0 and assign ownership
# of it to a non-root user.  The same goes for tun0 below.

tap = TuntapPort(b"tap0", EthernetProtocol(), reactor=reactor)
tap.startListening()
```

```
tun = TuntapPort(b"tun0", RawUDPProtocol(), reactor=reactor)
tun.startListening()
```

In the above example two protocols are attached to the network: one to a tap device and the other to a tun device. The `EthernetProtocol` used in this example is a very simple implementation of `IEthernetProtocol` which does nothing more than dispatch to some other protocol based on the protocol found in the header of each ethernet frame it receives. `RawUDPProtocol` is similar - it dispatches to other protocols based on the UDP port of IP datagrams it received. This example won't do anything since no application protocols have been added to either the `EthernetProtocol` or `RawUDPProtocol` instances (not to mention the reactor isn't being started). However, it should give you some idea of how tun/tap functionality fits into a Twisted application.

By the behaviors of these two protocols you can see the primary difference between tap and tun devices. The lower level of the two, tap devices, is hooked in to the network stack at the ethernet layer. When a `TuntapPort` is associated with a tap device, it delivers whole ethernet frames to its protocol. The higher level version, tun devices, strips off the ethernet layer before delivering data to the application. This means that a `TuntapPort` associated with a tun device most commonly delivers IP datagrams to its protocol (though if your network is being used to convey non-IP datagrams then it may deliver those instead).

Both `IEthernetProtocol` and `IRawSocketProtocol` are similar to twisted.internet.protocol.DatagramProtocol . Datagrams, either ethernet or otherwise, are delivered to the protocol's `datagramReceived` method. Conversely the protocol is associated with a transport with a `write` method that accepts datagrams for injection into the network.

You can see an example of some of this functionality in the `../examples/pairudp.py` example.

Twisted Pair: Device Configuration

Twisted Pair's Test Suite

Certain system configuration is required before the full Twisted Pair test suite can be run. Without this setup the test suite will lack the permission necessary to access tap and tun devices. Some tests will still run but the integration tests which verify Twisted Pair can successfully read from and write to real devices will be skipped.

The following shell script creates two tun devices and two tap devices and grants permission to use them to whatever user the shell script is run as. Run it to configure your system so that you can perform a complete run of the Twisted Pair test suite.

```
# Needs to be short enough so that with prefix and suffix, fits into 16 bytes
IDENTIFIER="twtest"

# A tap device without protocol information
sudo ip tuntap add dev tap-${IDENTIFIER} mode tap user $(id -u -n) group $(id -g -n)
sudo ip link set up dev tap-${IDENTIFIER}
sudo ip addr add 172.16.0.1/24 dev tap-${IDENTIFIER}
sudo ip neigh add 172.16.0.2 lladdr de:ad:be:ef:ca:fe dev tap-${IDENTIFIER}

# A tap device with protocol information
sudo ip tuntap add dev tap-${IDENTIFIER}-pi mode tap user $(id -u -n) group $(id -g -
↪n) pi
sudo ip link set up dev tap-${IDENTIFIER}-pi
sudo ip addr add 172.16.1.1/24 dev tap-${IDENTIFIER}-pi
sudo ip neigh add 172.16.1.2 lladdr de:ad:ca:fe:be:ef dev tap-${IDENTIFIER}-pi

# A tun device without protocol information
sudo ip tuntap add dev tun-${IDENTIFIER} mode tun user $(id -u -n) group $(id -g -n)
sudo ip link set up dev tun-${IDENTIFIER}
```

```
# A tun device with protocol information
sudo ip tuntap add dev tun-${IDENTIFIER}-pi mode tun user $(id -u -n) group $(id -g -
↪n) pi
sudo ip link set up dev tun-${IDENTIFIER}-pi
```

There are two things to keep in mind about this configuration. First, it uses addresses from the 172.16.0.0/12 private use range. If your network is configured to use these already then running the script may cause problems for your network. These addresses are hard-coded into the Twisted Pair test suite so this problem is not easily avoided. Second, the changes are not persistent across reboots. If you want this network configuration to be available even after a reboot you will need to integrate the above into your system's init scripts somehow (the details of this for different systems is beyond the scope of this document).

Certain platforms may also require a modification to their firewall rules in order to allow the traffic the test suite wants to transmit. Adding firewall rules which allowing traffic destined for the addresses used by the test suite should address this problem. If you encounter timeouts when running the Twisted Pair test suite then this may apply to you. For example, to configure an iptables firewall to allow this traffic:

```
iptables -I INPUT --dest 172.16.1.1 -j ACCEPT
iptables -I INPUT --dest 172.16.2.1 -j ACCEPT
```

- Twisted Pair Documentation
  - *Twisted Pair: Tunnels And Network Taps*
  - *Twisted Pair: Device Configuration*

Examples

Miscellaneous

- `pairudp.py` - UDP implemented with a TUN/TAP device
- *Developer guides*: documentation on using Twisted Pair to develop your own applications
- *Code Examples*: short code examples using Twisted Pair

CHAPTER 7

Twisted Web

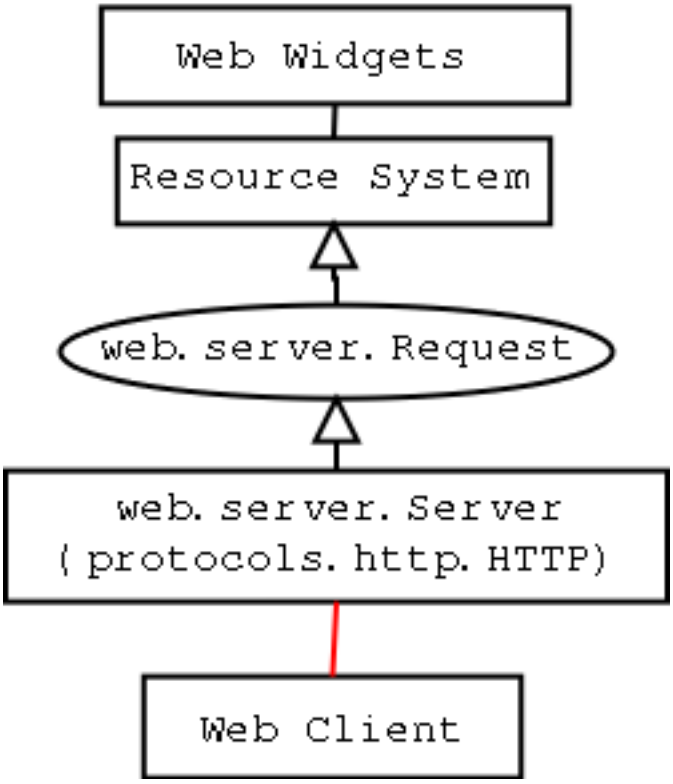
Developer Guides

Overview of Twisted Web

Introduction

Twisted Web is a web application server written in pure Python, with APIs at multiple levels of abstraction to facilitate different kinds of web programming.

Twisted Web's Structure



When the Web Server receives a request from a Client, it creates a Request object and passes it on to the Resource system. The Resource system dispatches to the appropriate Resource object based on what path was requested by the client. The Resource is asked to render itself, and the result is returned to the client.

Resources

Resources are the lowest-level abstraction for applications in the Twisted web server. Each Resource is a 1:1 mapping with a path that is requested: you can think of a Resource as a single "page" to be rendered. The interface for making Resources is very simple; they must have a method named `render` which takes a single argument, which is the Request object (an instance of twisted.web.server.Request). This render method must return a string, which will be returned to the web browser making the request. Alternatively, they can return a special constant, twisted.web.server.NOT_DONE_YET , which tells the web server not to close the connection; you must then use `request.write(data)` to render the page, and call `request.finish()` whenever you're done.

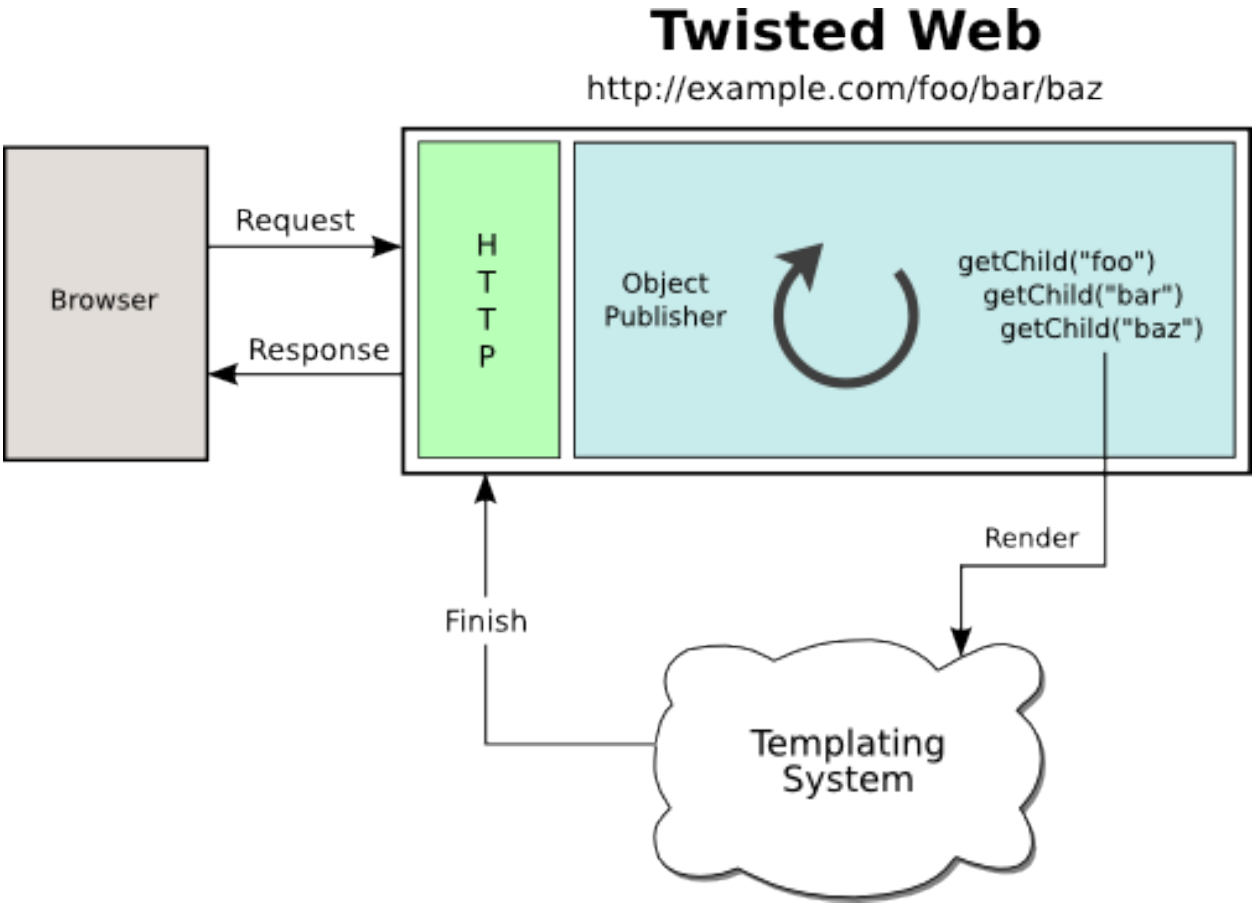
Web programming with Twisted Web

Web programmers seeking a higher level abstraction than the Resource system should look at Nevow . Nevow is based on ideas previously developed in Twisted, but is now maintained outside of Twisted to easy development and release cycle pressures.

Configuring and Using the Twisted Web Server

Twisted Web Development

Twisted Web serves Python objects that implement the interface IResource.



Main Concepts

- *Site Objects* are responsible for creating `HTTPChannel` instances to parse the HTTP request, and begin the object lookup process. They contain the root Resource, the resource which represents the URL `/` on the site.
- *Resource* objects represent a single URL segment. The IResource interface describes the methods a Resource object must implement in order to participate in the object publishing process.
- *Resource trees* are arrangements of Resource objects into a Resource tree. Starting at the root Resource object, the tree of Resource objects defines the URLs which will be valid.
- *.rpy scripts* are python scripts which the twisted.web static file server will execute, much like a CGI. However, unlike CGI they must create a Resource object which will be rendered when the URL is visited.
- *Resource rendering* occurs when Twisted Web locates a leaf Resource object. A Resource can either return an html string or write to the request object.
- *Session* objects allow you to store information across multiple requests. Each individual browser using the system has a unique Session instance.

The Twisted Web server is started through the Twisted Daemonizer, as in:

```
% twistd web
```

Site Objects

Site objects serve as the glue between a port to listen for HTTP requests on, and a root Resource object.

When using `twistd -n web --path /foo/bar/baz`, a Site object is created with a root Resource that serves files out of the given path.

You can also create a `Site` instance by hand, passing it a `Resource` object which will serve as the root of the site:

```
from twisted.web import server, resource
from twisted.internet import reactor, endpoints

class Simple(resource.Resource):
    isLeaf = True
    def render_GET(self, request):
        return "<html>Hello, world!</html>"

site = server.Site(Simple())
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8080)
endpoint.listen(site)
reactor.run()
```

Resource objects

`Resource` objects represent a single URL segment of a site. During URL parsing, `getChild` is called on the current `Resource` to produce the next `Resource` object.

When the leaf Resource is reached, either because there were no more URL segments or a Resource had isLeaf set to True, the leaf Resource is rendered by calling `render(request)`. See "Resource Rendering" below for more about this.

During the Resource location process, the URL segments which have already been processed and those which have not yet been processed are available in `request.prepath` and `request.postpath`.

A Resource can know where it is in the URL tree by looking at `request.prepath`, a list of URL segment strings.

A Resource can know which path segments will be processed after it by looking at `request.postpath`.

If the URL ends in a slash, for example `http://example.com/foo/bar/`, the final URL segment will be an empty string. Resources can thus know if they were requested with or without a final slash.

Here is a simple Resource object:

```
from twisted.web.resource import Resource

class Hello(Resource):
    isLeaf = True
    def getChild(self, name, request):
        if name == '':
            return self
        return Resource.getChild(self, name, request)

    def render_GET(self, request):
        return "Hello, world! I am located at %r." % (request.prepath,)
```

```
resource = Hello()
```

Resource Trees

Resources can be arranged in trees using `putChild`. `putChild` puts a Resource instance into another Resource instance, making it available at the given path segment name:

```
root = Hello()
root.putChild('fred', Hello())
root.putChild('bob', Hello())
```

If this root resource is served as the root of a Site instance, the following URLs will all be valid:

- `http://example.com/`
- `http://example.com/fred`
- `http://example.com/bob`
- `http://example.com/fred/`
- `http://example.com/bob/`

.rpy scripts

Files with the extension `.rpy` are python scripts which, when placed in a directory served by Twisted Web, will be executed when visited through the web.

An `.rpy` script must define a variable, `resource`, which is the Resource object that will render the request.

`.rpy` files are very convenient for rapid development and prototyping. Since they are executed on every web request, defining a Resource subclass in an `.rpy` will make viewing the results of changes to your class visible simply by refreshing the page:

```
from twisted.web.resource import Resource

class MyResource(Resource):
    def render_GET(self, request):
        return "<html>Hello, world!</html>"

resource = MyResource()
```

However, it is often a better idea to define Resource subclasses in Python modules. In order for changes in modules to be visible, you must either restart the Python process, or reload the module:

```
import myresource

## Comment out this line when finished debugging
reload(myresource)

resource = myresource.MyResource()
```

Creating a Twisted Web server which serves a directory is easy:

```
% twistd -n web --path /Users/dsp/Sites
```

Resource rendering

Resource rendering occurs when Twisted Web locates a leaf Resource object to handle a web request. A Resource's `render` method may do various things to produce output which will be sent back to the browser:

- Return a string
- Call `request.write("stuff")` as many times as desired, then call `request.finish()` and return `server.NOT_DONE_YET` (This is deceptive, since you are in fact done with the request, but is the correct way to do this)
- Request a `Deferred` , return `server.NOT_DONE_YET` , and call `request.write("stuff")` and `request.finish()` later, in a callback on the `Deferred` .

The Resource class, which is usually what one's Resource classes subclass, has a convenient default implementation of `render` . It will call a method named `self.render_METHOD` where "METHOD" is whatever HTTP method was used to request this resource. Examples: request_GET, request_POST, request_HEAD, and so on. It is recommended that you have your resource classes subclass Resource and implement `render_METHOD` methods as opposed to `render` itself. Note that for certain resources, `request_POST = request_GET` may be desirable in case one wants to process arguments passed to the resource regardless of whether they used GET (`?foo=bar&baz=quux` , and so forth) or POST.

Request encoders

When using a Resource , one can specify wrap it using a EncodingResourceWrapper and passing a list of encoder factories. The encoder factories are called when a request is processed and potentially return an encoder. By default twisted provides GzipEncoderFactory which manages standard gzip compression. You can use it this way:

```
from twisted.web.server import Site, GzipEncoderFactory
from twisted.web.resource import Resource, EncodingResourceWrapper
from twisted.internet import reactor, endpoints

class Simple(Resource):
    isLeaf = True
    def render_GET(self, request):
        return "<html>Hello, world!</html>"

resource = Simple()
wrapped = EncodingResourceWrapper(resource, [GzipEncoderFactory()])
site = Site(wrapped)
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8080)
endpoint.listen(site)
reactor.run()
```

Using compression on SSL served resources where the user can influence the content can lead to information leak, so be careful which resources use request encoders.

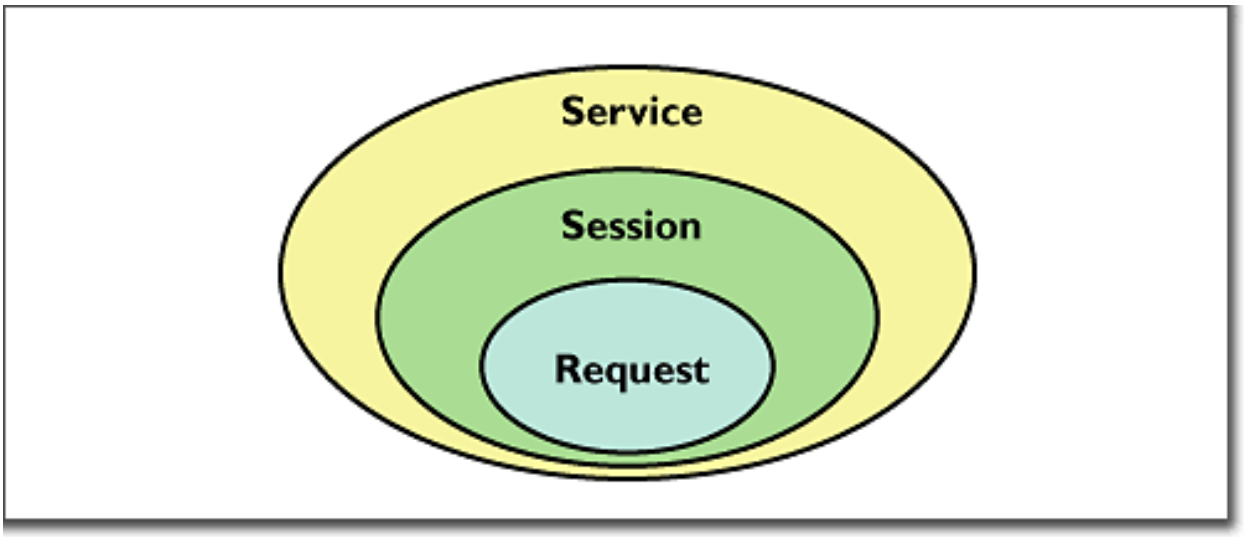
Note that only encoder can be used per request: the first encoder factory returning an object will be used, so the order in which they are specified matters.

Session

HTTP is a stateless protocol; every request-response is treated as an individual unit, distinguishable from any other request only by the URL requested. With the advent of Cookies in the mid nineties, dynamic web servers gained the ability to distinguish between requests coming from different *browser sessions* by sending a Cookie to a browser. The

browser then sends this cookie whenever it makes a request to a web server, allowing the server to track which requests come from which browser session.

Twisted Web provides an abstraction of this browser-tracking behavior called the *Session object* . Calling `request.getSession()` checks to see if a session cookie has been set; if not, it creates a unique session id, creates a Session object, stores it in the Site, and returns it. If a session object already exists, the same session object is returned. In this way, you can store data specific to the session in the session object.



Proxies and reverse proxies

A proxy is a general term for a server that functions as an intermediary between clients and other servers.

Twisted supports two main proxy variants: a Proxy and a ReverseProxy .

Proxy

A proxy forwards requests made by a client to a destination server. Proxies typically sit on the internal network for a client or out on the internet, and have many uses, including caching, packet filtering, auditing, and circumventing local access restrictions to web content.

Here is an example of a simple but complete web proxy:

```
from twisted.web import proxy, http
from twisted.internet import reactor, endpoints

class ProxyFactory(http.HTTPFactory):
    def buildProtocol(self, addr):
        return proxy.Proxy()

endpoint = endpoints.TCP4ServerEndpoint(reactor, 8080)
endpoint.listen(ProxyFactory())
reactor.run()
```

With this proxy running, you can configure your web browser to use `localhost:8080` as a proxy. After doing so, when browsing the web all requests will go through this proxy.

Proxy inherits from http.HTTPChannel . Each client request to the proxy generates a ProxyRequest from the proxy to the destination server on behalf of the client. `ProxyRequest` uses a ProxyClientFactory to create an instance of the

ProxyClient protocol for the connection. `ProxyClient` inherits from http.HTTPClient . Subclass `ProxyRequest` to customize the way requests are processed or logged.

ReverseProxyResource

A reverse proxy retrieves resources from other servers on behalf of a client. Reverse proxies typically sit inside the server's internal network and are used for caching, application firewalls, and load balancing.

Here is an example of a basic reverse proxy:

```
from twisted.internet import reactor, endpoints
from twisted.web import proxy, server

site = server.Site(proxy.ReverseProxyResource('www.yahoo.com', 80, ''))
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8080)
endpoint.listen(site)
reactor.run()
```

With this reverse proxy running locally, you can visit `http://localhost:8080` in your web browser, and the reverse proxy will proxy your connection to `www.yahoo.com`.

In this example we use `server.Site` to serve a `ReverseProxyResource` directly. There is also a `ReverseProxy` family of classes in `twisted.web.proxy` mirroring those of the `Proxy` family:

Like `Proxy` , ReverseProxy inherits from `http.HTTPChannel` . Each client request to the reverse proxy generates a ReverseProxyRequest to the destination server. Like `ProxyRequest` , ReverseProxyRequest uses a ProxyClient-Factory to create an instance of the ProxyClient protocol for the connection.

Additional examples of proxies and reverse proxies can be found in the Twisted web examples

Advanced Configuration

Non-trivial configurations of Twisted Web are achieved with Python configuration files. This is a Python snippet which builds up a variable called application. Usually, the `twisted.application.strports.service` function will be used to build a service instance that will be used to make the application listen on a TCP port (80, in case direct web serving is desired), with the listener being a twisted.web.server.Site . The resulting file can then be run with `twistd -y` . Alternatively a reactor object can be used directly to make a runnable script.

The `Site` will wrap a `Resource` object – the root.

```
from twisted.application import internet, service, strports
from twisted.web import static, server

root = static.File("/var/www/htdocs")
application = service.Application('web')
site = server.Site(root)
sc = service.IServiceCollection(application)
i = strports.service("tcp:80", site)
i.setServiceParent(sc)
```

Most advanced configurations will be in the form of tweaking the root resource object.

Adding Children

Usually, the root's children will be based on the filesystem's contents. It is possible to override the filesystem by explicit `putChild` methods.

Here are two examples. The first one adds a `/doc` child to serve the documentation of the installed packages, while the second one adds a `cgi-bin` directory for CGI scripts.

```
from twisted.internet import reactor, endpoints
from twisted.web import static, server

root = static.File("/var/www/htdocs")
root.putChild("doc", static.File("/usr/share/doc"))
endpoint = endpoints.TCP4ServerEndpoint(reactor, 80)
endpoint.listen(server.Site(root))
reactor.run()
```

```
from twisted.internet import reactor, endpoints
from twisted.web import static, server, twcgi

root = static.File("/var/www/htdocs")
root.putChild("cgi-bin", twcgi.CGIDirectory("/var/www/cgi-bin"))
endpoint = endpoints.TCP4ServerEndpoint(reactor, 80)
endpoint.listen(server.Site(root))
reactor.run()
```

Modifying File Resources

`File` resources, be they root object or children thereof, have two important attributes that often need to be modified: `indexNames` and `processors` . `indexNames` determines which files are treated as "index files" – served up when a directory is rendered. `processors` determine how certain file extensions are treated.

Here is an example for both, creating a site where all `.rpy` extensions are Resource Scripts, and which renders directories by searching for a `index.rpy` file.

```
from twisted.application import internet, service, strports
from twisted.web import static, server, script

root = static.File("/var/www/htdocs")
root.indexNames=['index.rpy']
root.processors = {'.rpy': script.ResourceScript}
application = service.Application('web')
sc = service.IServiceCollection(application)
site = server.Site(root)
i = strports.service("tcp:80", site)
i.setServiceParent(sc)
```

`File` objects also have a method called `ignoreExt` . This method can be used to give extension-less URLs to users, so that implementation is hidden. Here is an example:

```
from twisted.application import internet, service, strports
from twisted.web import static, server, script

root = static.File("/var/www/htdocs")
root.ignoreExt(".rpy")
root.processors = {'.rpy': script.ResourceScript}
application = service.Application('web')
sc = service.IServiceCollection(application)
site = server.Site(root)
i = strports.service("tcp:80", site)
i.setServiceParent(sc)
```

Now, a URL such as `/foo` might be served from a Resource Script called `foo.rpy` , if no file by the name of `foo` exists.

`File` objects will try to automatically determine the Content-Type and Content-Encoding headers. There is a small set of known mime types and encodings which augment the default mime types provided by the Python standard library *mimetypes*. You can always modify the content type and encoding mappings by manipulating the instance variables.

For example to recognize WOFF File Format 2.0 and set the right Content-Type header you can modify the *contentTypes* member of an instance:

```
.. code-block:: python
```

from twisted.application import internet, service, strports from twisted.web import static, server, script

root = static.File("/srv/fonts")

root.contentTypes[".woff2"] = "application/font-woff2"

application = service.Application('web') sc = service.IServiceCollection(application) site = server.Site(root) i = strports.service("tcp:80", site) i.setServiceParent(sc)

Virtual Hosts

Virtual hosting is done via a special resource, that should be used as the root resource – `NameVirtualHost` . `NameVirtualHost` has an attribute named `default` , which holds the default website. If a different root for some other name is desired, the `addHost` method should be called.

```
from twisted.application import internet, service, strports
from twisted.web import static, server, vhost, script

root = vhost.NameVirtualHost()

# Add a default -- htdocs
root.default=static.File("/var/www/htdocs")

# Add a simple virtual host -- foo.com
root.addHost("foo.com", static.File("/var/www/foo"))

# Add a simple virtual host -- bar.com
root.addHost("bar.com", static.File("/var/www/bar"))

# The "baz" people want to use Resource Scripts in their web site
baz = static.File("/var/www/baz")
baz.processors = {'.rpy': script.ResourceScript}
baz.ignoreExt('.rpy')
root.addHost('baz', baz)

application = service.Application('web')
sc = service.IServiceCollection(application)
site = server.Site(root)
i = strports.service("tcp:80", site)
i.setServiceParent(sc)
```

Advanced Techniques

Since the configuration is a Python snippet, it is possible to use the full power of Python. Here are some simple examples:

```
# No need for configuration of virtual hosts -- just make sure
# a directory /var/vhosts/<vhost name> exists:
from twisted.web import vhost, static, server
from twisted.application import internet, service, strports

root = vhost.NameVirtualHost()
root.default = static.File("/var/www/htdocs")
for dir in os.listdir("/var/vhosts"):
    root.addHost(dir, static.File(os.path.join("/var/vhosts", dir)))

application = service.Application('web')
sc = service.IServiceCollection(application)
site = server.Site(root)
i = strports.service("tcp:80", site)
i.setServiceParent(sc)
```

```
# Determine ports we listen on based on a file with numbers:
from twisted.web import vhost, static, server
from twisted.application import internet, service

root = static.File("/var/www/htdocs")

site = server.Site(root)
application = service.Application('web')
serviceCollection = service.IServiceCollection(application)

with open("/etc/web/ports") as f:
    for num in map(int, f.read().split()):
        serviceCollection.addCollection(
            strports.service("tcp:%d" % num, site)
        )
```

Running a Twisted Web Server

In many cases, you'll end up repeating common usage patterns of twisted.web. In those cases you'll probably want to use Twisted's pre-configured web server setup.

The easiest way to run a Twisted Web server is with the Twisted Daemonizer. For example, this command will run a web server which serves static files from a particular directory:

```
% twistd web --path /path/to/web/content
```

If you just want to serve content from your own home directory, the following will do:

```
% twistd web --path ~/public_html/
```

You can stop the server at any time by going back to the directory you started it in and running the command:

```
% kill `cat twistd.pid`
```

Some other configuration options are available as well:

- `--port` : Specify the port for the web server to listen on. This defaults to 8080.
- `--logfile` : Specify the path to the log file.

- `--add-header`: Specify additional headers to be served with every response. These are formatted like `--add-header "HeaderName: HeaderValue"`.

The full set of options that are available can be seen with:

```
% twistd web --help
```

Serving Flat HTML

Twisted Web serves flat HTML files just as it does any other flat file.

Resource Scripts

A Resource script is a Python file ending with the extension `.rpy` , which is required to create an instance of a (subclass of a) twisted.web.resource.Resource .

Resource scripts have 3 special variables:

- `__file__` : The name of the .rpy file, including the full path. This variable is automatically defined and present within the namespace.
- `registry` : An object of class static.Registry . It can be used to access and set persistent data keyed by a class.
- `resource` : The variable which must be defined by the script and set to the resource instance that will be used to render the page.

A very simple Resource Script might look like:

```
from twisted.web import resource
class MyGreatResource(resource.Resource):
    def render_GET(self, request):
        return "<html>foo</html>"

resource = MyGreatResource()
```

A slightly more complicated resource script, which accesses some persistent data, might look like:

```
from twisted.web import resource
from SillyWeb import Counter

counter = registry.getComponent(Counter)
if not counter:
   registry.setComponent(Counter, Counter())
counter = registry.getComponent(Counter)

class MyResource(resource.Resource):
    def render_GET(self, request):
        counter.increment()
        return "you are visitor %d" % counter.getValue()

resource = MyResource()
```

This is assuming you have the `SillyWeb.Counter` module, implemented something like the following:

```
class Counter:

    def __init__(self):
```

```
        self.value = 0

    def increment(self):
        self.value += 1

    def getValue(self):
        return self.value
```

Web UIs

The Nevow framework, available as part of the Quotient project, is an advanced system for giving Web UIs to your application. Nevow uses Twisted Web but is not itself part of Twisted.

Spreadable Web Servers

One of the most interesting applications of Twisted Web is the distributed webserver; multiple servers can all answer requests on the same port, using the twisted.spread package for "spreadable" computing. In two different directories, run the commands:

```
% twistd web --user
% twistd web --personal [other options, if you desire]
```

Once you're running both of these instances, go to `http://localhost:8080/your_username.twistd/` – you will see the front page from the server you created with the `--personal` option. What's happening here is that the request you've sent is being relayed from the central (User) server to your own (Personal) server, over a PB connection. This technique can be highly useful for small "community" sites; using the code that makes this demo work, you can connect one HTTP port to multiple resources running with different permissions on the same machine, on different local machines, or even over the internet to a remote site.

By default, a personal server listens on a UNIX socket in the owner's home directory. The `--port` option can be used to make it listen on a different address, such as a TCP or SSL server or on a UNIX server in a different location. If you use this option to make a personal server listen on a different address, the central (User) server won't be able to find it, but a custom server which uses the same APIs as the central server might. Another use of the `--port` option is to make the UNIX server robust against system crashes. If the server crashes and the UNIX socket is left on the filesystem, the personal server will not be able to restart until it is removed. However, if `--port unix:/home/username/.twistd-web-pb:wantPID=1` is supplied when creating the personal server, then a lockfile will be used to keep track of whether the server socket is in use and automatically delete it when it is not.

Serving PHP/Perl/CGI

Everything related to CGI is located in the `twisted.web.twcgi` , and it's here you'll find the classes that you need to subclass in order to support the language of your (or somebody elses) taste. You'll also need to create your own kind of resource if you are using a non-unix operating system (such as Windows), or if the default resources has wrong pathnames to the parsers.

The following snippet is a .rpy that serves perl-files. Look at `twisted.web.twcgi` for more examples regarding twisted.web and CGI.

```
from twisted.web import static, twcgi

class PerlScript(twcgi.FilteredScript):
    filter = '/usr/bin/perl' # Points to the perl parser
```

```
resource = static.File("/perlsite") # Points to the perl website
resource.processors = {".pl": PerlScript} # Files that end with .pl will be
                                          # processed by PerlScript
resource.indexNames = ['index.pl']
```

Serving WSGI Applications

WSGI is the Web Server Gateway Interface. It is a specification for web servers and application servers to communicate with Python web applications. All modern Python web frameworks support the WSGI interface.

The easiest way to get started with WSGI application is to use the twistd command:

```
% twistd -n web --wsgi=helloworld.application
```

This assumes that you have a WSGI application called application in your helloworld module/package, which might look like this:

```
def application(environ, start_response):
    """Basic WSGI Application"""
    start_response('200 OK', [('Content-type','text/plain')])
    return ['Hello World!']
```

The above setup will be suitable for many applications where all that is needed is to server the WSGI application at the site's root. However, for greater control, Twisted provides support for using WSGI applications as resources `twisted.web.wsgi.WSGIResource`.

Here is an example of a WSGI application being served as the root resource for a site, in the following tac file:

```
from twisted.web import server
from twisted.web.wsgi import WSGIResource
from twisted.python.threadpool import ThreadPool
from twisted.internet import reactor
from twisted.application import service, strports

# Create and start a thread pool,
wsgiThreadPool = ThreadPool()
wsgiThreadPool.start()

# ensuring that it will be stopped when the reactor shuts down
reactor.addSystemEventTrigger('after', 'shutdown', wsgiThreadPool.stop)

def application(environ, start_response):
    """A basic WSGI application"""
    start_response('200 OK', [('Content-type','text/plain')])
    return ['Hello World!']

# Create the WSGI resource
wsgiAppAsResource = WSGIResource(reactor, wsgiThreadPool, application)

# Hooks for twistd
application = service.Application('Twisted.web.wsgi Hello World Example')
server = strports.service('tcp:8080', server.Site(wsgiAppAsResource))
server.setServiceParent(application)
```

This can then be run like any other .tac file:

```
% twistd -ny myapp.tac
```

Because of the synchronous nature of WSGI, each application call (for each request) is called within a thread, and the result is written back to the web server. For this, a `twisted.python.threadpool.ThreadPool` instance is used.

Using VHostMonster

It is common to use one server (for example, Apache) on a site with multiple names which then uses reverse proxy (in Apache, via `mod_proxy`) to different internal web servers, possibly on different machines. However, naive configuration causes miscommunication: the internal server firmly believes it is running on "internal-name:port" , and will generate URLs to that effect, which will be completely wrong when received by the client.

While Apache has the ProxyPassReverse directive, it is really a hack and is nowhere near comprehensive enough. Instead, the recommended practice in case the internal web server is Twisted Web is to use VHostMonster.

From the Twisted side, using VHostMonster is easy: just drop a file named (for example) `vhost.rpy` containing the following:

```
from twisted.web import vhost
resource = vhost.VHostMonsterResource()
```

Make sure the web server is configured with the correct processors for the `rpy` extensions (the web server `twistd web --path` generates by default is so configured).

From the Apache side, instead of using the following ProxyPass directive:

```
<VirtualHost ip-addr>
ProxyPass / http://localhost:8538/
ServerName example.com
</VirtualHost>
```

Use the following directive:

```
<VirtualHost ip-addr>
ProxyPass / http://localhost:8538/vhost.rpy/http/example.com:80/
ServerName example.com
</VirtualHost>
```

Here is an example for Twisted Web's reverse proxy:

```
from twisted.application import internet, service, strports
from twisted.web import proxy, server, vhost
vhostName = 'example.com'
reverseProxy = proxy.ReverseProxyResource('internal', 8538,
                                          '/vhost.rpy/http/'+vhostName+'/')
root = vhost.NameVirtualHost()
root.addHost(vhostName, reverseProxy)
site = server.Site(root)
application = service.Application('web-proxy')
sc = service.IServiceCollection(application)
i = strports.service("tcp:80", site)
i.setServiceParent(sc)
```

Rewriting URLs

Sometimes it is convenient to modify the content of the Request object before passing it on. Because this is most often used to rewrite either the URL, the similarity to Apache's `mod_rewrite` has inspired the twisted.web.rewrite module. Using this module is done via wrapping a resource with a twisted.web.rewrite.RewriterResource which then has rewrite rules. Rewrite rules are functions which accept a request object, and possible modify it. After all rewrite rules run, the child resolution chain continues as if the wrapped resource, rather than the RewriterResource , was the child.

Here is an example, using the only rule currently supplied by Twisted itself:

```
default_root = rewrite.RewriterResource(default, rewrite.tildeToUsers)
```

This causes the URL `/~foo/bar.html` to be treated like `/users/foo/bar.html` . If done after setting default's `users` child to a distrib.UserDirectory , it gives a configuration similar to the classical configuration of web server, common since the first NCSA servers.

Knowing When We're Not Wanted

Sometimes it is useful to know when the other side has broken the connection. Here is an example which does that:

```
from twisted.web.resource import Resource
from twisted.web import server
from twisted.internet import reactor
from twisted.python.util import println


class ExampleResource(Resource):

    def render_GET(self, request):
        request.write("hello world")
        d = request.notifyFinish()
        d.addCallback(lambda _: println("finished normally"))
        d.addErrback(println, "error")
        reactor.callLater(10, request.finish)
        return server.NOT_DONE_YET

resource = ExampleResource()
```

This will allow us to run statistics on the log-file to see how many users are frustrated after merely 10 seconds.

As-Is Serving

Sometimes, you want to be able to send headers and status directly. While you can do this with a ResourceScript , an easier way is to use ASISProcessor . Use it by, for example, adding it as a processor for the `.asis` extension. Here is a sample file:

```
HTTP/1.0 200 OK
Content-Type: text/html

Hello world
```

Web Application Development

Code layout

The development of a Twisted Web application should be orthogonal to its deployment. This means is that if you are developing a web application, it should be a resource with children, and internal links. Some of the children might use Nevow , some might be resources manually using `.write` , and so on. Regardless, the code should be in a Python module, or package, *outside* the web tree.

You will probably want to test your application as you develop it. There are many ways to test, including dropping an `.rpy` which looks like:

```
from mypackage import toplevel
resource = toplevel.Resource(file="foo/bar", color="blue")
```

into a directory, and then running:

```
% twistd web --path=/directory
```

You can also write a Python script like:

```
#!/usr/bin/env python

from twisted.web import server
from twisted.internet import reactor, endpoints
from mypackage import toplevel

endpoint = endpoints.TCP4ServerEndpoint(reactor, 8080)
endpoint.listen(
    server.Site(toplevel.Resource(file="foo/bar", color="blue")))
reactor.run()
```

Web application deployment

Which one of these development strategies you use is not terribly important, since (and this is the important part) deployment is *orthogonal* . Later, when you want users to actually *use* your code, you should worry about what to do – or rather, don't. Users may have widely different needs. Some may want to run your code in a different process, so they'll use distributed web (twisted.web.distrib). Some may be using the `twisted-web` Debian package, and will drop in:

```
% cat > /etc/local.d/99addmypackage.py
from mypackage import toplevel
default.putChild("mypackage", toplevel.Resource(file="foo/bar", color="blue"))
^D
```

If you want to be friendly to your users, you can supply many examples in your package, like the above `.rpy` and the Debian-package drop-in. But the *ultimate* friendliness is to write a useful resource which does not have deployment assumptions built in.

Understanding resource scripts (`.rpy` files)

Twisted Web is not PHP – it has better tools for organizing code Python modules and packages, so use them. In PHP, the only tool for organizing code is a web page, which leads to silly things like PHP pages full of functions that other pages import, and so on. If you were to write your code this way with Twisted Web, you would do web development using many `.rpy` files, all importing some Python module. This is a *bad idea* – it mashes deployment with development, and makes sure your users will be *tied* to the file-system.

We have `.rpy` s because they are useful and necessary. But using them incorrectly leads to horribly unmaintainable applications. The best way to ensure you are using them correctly is to not use them at all, until you are on your *final* deployment stages. You should then find your `.rpy` files will be less than 10 lines, because you will not *have* more than 10 lines to write.

HTML Templating with twisted.web.template

A Very Quick Introduction To Templating In Python

HTML templating is the process of transforming a template document (one which describes style and structure, but does not itself include any content) into some HTML output which includes information about objects in your application. There are many, many libraries for doing this in Python: to name a few, jinja2 , django templates , and clearsilver . You can easily use any of these libraries in your Twisted Web application, either by running them as *WSGI applications* or by calling your preferred templating system's APIs to produce their output as strings, and then writing those strings to Request.write .

Before we begin explaining how to use it, I'd like to stress that you don't *need* to use Twisted's templating system if you prefer some other way to generate HTML. Use it if it suits your personal style or your application, but feel free to use other things. Twisted includes templating for its own use, because the `twisted.web` server needs to produce HTML in various places, and we didn't want to add another large dependency for that. Twisted is *not* in any way incompatible with other systems, so that has nothing to do with the fact that we use our own.

twisted.web.template - Why And How you Might Want to Use It

Twisted includes a templating system, twisted.web.template . This can be convenient for Twisted applications that want to produce some basic HTML for a web interface without an additional dependency.

`twisted.web.template` also includes support for Deferred s, so you can incrementally render the output of a page based on the results of Deferred s that your application has returned. This feature is fairly unique among templating libraries.

In twisted.web.template , templates are XHTML files which also contain a special namespace for indicating dynamic portions of the document. For example:

`template-1.xml`

```
<html xmlns:t="http://twistedmatrix.com/ns/twisted.web.template/0.1">
<body>
  <div t:render="header" />
  <div id="content">
    <p>Content goes here.</p>
  </div>
  <div t:render="footer" />
</body>
</html>
```

The basic unit of templating is twisted.web.template.Element . An Element is given a way of loading a bit of markup like the above example, and knows how to correlate `render` attributes within that markup to Python methods exposed with twisted.web.template.renderer :

`element_1.py`

```
from twisted.web.template import Element, renderer, XMLFile
from twisted.python.filepath import FilePath

class ExampleElement(Element):
```

```
    loader = XMLFile(FilePath('template-1.xml'))

    @renderer
    def header(self, request, tag):
        return tag('Header.')

    @renderer
    def footer(self, request, tag):
        return tag('Footer.')
```

In order to combine the two, we must render the element. For this simple example, we can use the flattenString API, which will convert a single template object - such as an Element - into a Deferred which fires with a single string, the HTML output of the rendering process.

render_1.py

```
from __future__ import print_function

from twisted.web.template import flattenString
from element_1 import ExampleElement
def renderDone(output):
    print(output)
flattenString(None, ExampleElement()).addCallback(renderDone)
```

This short program cheats a little bit; we know that there are no Deferred s in the template which require the reactor to eventually fire; therefore, we can simply add a callback which outputs the result. Also, none of the `renderer` functions require the `request` object, so it's acceptable to pass `None` through here. (The 'request' object here is used only to relay information about the rendering process to each renderer, so you may always use whatever object makes sense for your application. Note, however, that renderers from library code may require an IRequest .)

If you run it yourself, you can see that it produces the following output:

output-1.html

```
<html>
<body>
  <div>Header.</div>
  <div id="content">
    <p>Content goes here.</p>
  </div>
  <div>Footer.</div>
</body>
</html>
```

The third parameter to a renderer method is a Tag object which represents the XML element with the `t:render` attribute in the template. Calling a Tag adds children to the element in the DOM, which may be strings, more Tag s, or other renderables such as Element s. For example, to make the header and footer bold:

element_2.py

```
from twisted.web.template import Element, renderer, XMLFile, tags
from twisted.python.filepath import FilePath

class ExampleElement(Element):
    loader = XMLFile(FilePath('template-1.xml'))

    @renderer
    def header(self, request, tag):
        return tag(tags.b('Header.'))
```

```
    @renderer
    def footer(self, request, tag):
        return tag(tags.b('Footer.'))
```

Rendering this in a similar way to the first example would produce:

output-2.html

```
<html>
<body>
  <div><b>Header.</b></div>
  <div id="content">
    <p>Content goes here.</p>
  </div>
  <div><b>Footer.</b></div>
</body>
</html>
```

In addition to adding children, call syntax can be used to set attributes on a tag. For example, to change the `id` on the `div` while adding children:

element_3.py

```
from twisted.web.template import Element, renderer, XMLFile, tags
from twisted.python.filepath import FilePath

class ExampleElement(Element):
    loader = XMLFile(FilePath('template-1.xml'))

    @renderer
    def header(self, request, tag):
        return tag(tags.p('Header.'), id='header')

    @renderer
    def footer(self, request, tag):
        return tag(tags.p('Footer.'), id='footer')
```

And this would produce the following page:

output-3.html

```
<html>
<body>
  <div id="header"><p>Header.</p></div>
  <div id="content">
    <p>Content goes here.</p>
  </div>
  <div id="footer"><p>Footer.</p></div>
</body>
</html>
```

Calling a tag mutates it, it and returns the tag itself, so you can pass it forward and call it multiple times if you have multiple children or attributes to add to it. twisted.web.template also exposes some convenient objects for building more complex markup structures from within renderer methods in the `tags` object. In the examples above, we've only used `tags.p` and `tags.b` , but there should be a `tags.x` for each *x* which is a valid HTML tag. There may be some omissions, but if you find one, please feel free to file a bug.

Template Attributes

`t:attr` tags allow you to set HTML attributes (like `href` in an `<a href="...`) on an enclosing element.

Slots

`t:slot` tags allow you to specify "slots" which you can conveniently fill with multiple pieces of data straight from your Python program.

The following example demonstrates both `t:attr` and `t:slot` in action. Here we have a layout which displays a person's profile on your snazzy new Twisted-powered social networking site. We use the `t:attr` tag to drop in the "src" attribute on the profile picture, where the actual value of src attribute gets specified by a `t:slot` tag *within* the `t:attr` tag. Confused? It should make more sense when you see the code:

`slots-attributes-1.xml`

```
<div xmlns:t="http://twistedmatrix.com/ns/twisted.web.template/0.1"
    t:render="person_profile"
    class="profile">
<img><t:attr name="src"><t:slot name="profile_image_url" /></t:attr></img>
<p><t:slot name="person_name" /></p>
</div>
```

`slots_attributes_1.py`

```
from twisted.web.template import Element, renderer, XMLFile
from twisted.python.filepath import FilePath

class ExampleElement(Element):
    loader = XMLFile(FilePath('slots-attributes-1.xml'))

    @renderer
    def person_profile(self, request, tag):
        # Note how convenient it is to pass these attributes in!
        tag.fillSlots(person_name='Luke',
                      profile_image_url='http://example.com/user.png')
        return tag
```

`slots-attributes-output.html`

```
<div class="profile">
<img src="http://example.com/user.png" />
<p>Luke</p>
</div>
```

Iteration

Often, you will have a sequence of things, and want to render each of them, repeating a part of the template for each one. This can be done by cloning `tag` in your renderer:

`iteration-1.xml`

```
<ul xmlns:t="http://twistedmatrix.com/ns/twisted.web.template/0.1">
    <li t:render="widgets"><t:slot name="widgetName"/></li>
</ul>
```

iteration-1.py

```
from __future__ import print_function

from twisted.web.template import Element, renderer, XMLFile, flattenString
from twisted.python.filepath import FilePath

class WidgetsElement(Element):
    loader = XMLFile(FilePath('iteration-1.xml'))

    widgetData = ['gadget', 'contraption', 'gizmo', 'doohickey']

    @renderer
    def widgets(self, request, tag):
        for widget in self.widgetData:
            yield tag.clone().fillSlots(widgetName=widget)

def printResult(result):
    print(result)

flattenString(None, WidgetsElement()).addCallback(printResult)
```

iteration-output-1.xml

```
<ul>
    <li>gadget</li><li>contraption</li><li>gizmo</li><li>doohickey</li>
</ul>
```

This renderer works because a renderer can return anything that can be rendered, not just `tag` . In this case, we define a generator, which returns a thing that is iterable. We also could have returned a `list` . Anything that is iterable will be rendered by twisted.web.template rendering each item in it. In this case, each item is a copy of the tag the renderer received, each filled with the name of a widget.

Sub-views

Another common pattern is to delegate the rendering logic for a small part of the page to a separate `Element` . For example, the widgets from the iteration example above might be more complicated to render. You can define an `Element` subclass which can render a single widget. The renderer method on the container can then yield instances of this new `Element` subclass.

subviews-1.xml

```
<ul xmlns:t="http://twistedmatrix.com/ns/twisted.web.template/0.1">
    <li t:render="widgets"><span t:render="name" /></li>
</ul>
```

subviews-1.py

```
from __future__ import print_function

from twisted.web.template import (
    XMLFile, TagLoader, Element, renderer, flattenString)
from twisted.python.filepath import FilePath

class WidgetsElement(Element):
    loader = XMLFile(FilePath('subviews-1.xml'))
```

```
    widgetData = ['gadget', 'contraption', 'gizmo', 'doohickey']

    @renderer
    def widgets(self, request, tag):
        for widget in self.widgetData:
            yield WidgetElement(TagLoader(tag), widget)

class WidgetElement(Element):
    def __init__(self, loader, name):
        Element.__init__(self, loader)
        self._name = name

    @renderer
    def name(self, request, tag):
        return tag(self._name)

def printResult(result):
    print(result)

flattenString(None, WidgetsElement()).addCallback(printResult)
```

subviews-output-1.xml

```
<ul>
      <li><span>gadget</span></li><li><span>contraption</span></li><li><span>gizmo</
↪span></li><li><span>doohickey</span></li>
</ul>
```

`TagLoader` lets the portion of the overall template related to widgets be re-used for `WidgetElement` , which is otherwise a normal `Element` subclass not much different from `WidgetsElement` . Notice that the *name* renderer on the `span` tag in this template is satisfied from `WidgetElement` , not `WidgetsElement` .

Transparent

Note how renderers, slots and attributes require you to specify a renderer on some outer HTML element. What if you don't want to be forced to add an element to your DOM just to drop some content into it? Maybe it messes with your layout, and you can't get it to work in IE with that extra `div` tag? Perhaps you need `t:transparent` , which allows you to drop some content in without any surrounding "container" tag. For example:

transparent-1.xml

```
<div xmlns:t="http://twistedmatrix.com/ns/twisted.web.template/0.1">
<!-- layout decision - these things need to be *siblings* -->
<t:transparent t:render="renderer1" />
<t:transparent t:render="renderer2" />
</div>
```

transparent_element.py

```
from twisted.web.template import Element, renderer, XMLFile
from twisted.python.filepath import FilePath

class ExampleElement(Element):
    loader = XMLFile(FilePath('transparent-1.xml'))
```

```
    @renderer
    def renderer1(self, request, tag):
        return tag("hello")

    @renderer
    def renderer2(self, request, tag):
        return tag("world")
```

transparent-output.html

```
<div>
<!-- layout decision - these things need to be *siblings* -->
hello
world
</div>
```

Quoting

twisted.web.template will quote any strings that place into the DOM. This provides protection against XSS attacks , in addition to just generally making it easy to put arbitrary strings onto a web page, without worrying about what they might have in them. This can easily be demonstrated with an element using the same template from our earlier examples. Here's an element that returns some "special" characters in HTML ('<', '>', and '"', which is special in attribute values):

quoting_element.py

```
from twisted.web.template import Element, renderer, XMLFile
from twisted.python.filepath import FilePath

class ExampleElement(Element):
    loader = XMLFile(FilePath('template-1.xml'))

    @renderer
    def header(self, request, tag):
        return tag('<<<Header>>>!')

    @renderer
    def footer(self, request, tag):
        return tag('>>>"Footer!"<<<', id='<"fun">')
```

Note that they are all safely quoted in the output, and will appear in a web browser just as you returned them from your Python method:

quoting-output.html

```
<html>
<body>
  <div>&lt;&lt;&lt;Header&gt;&gt;&gt;!</div>
  <div id="content">
    <p>Content goes here.</p>
  </div>
  <div id="&lt;&quot;fun&quot;&gt;">&gt;&gt;&gt;"Footer!"&lt;&lt;&lt;</div>
</body>
</html>
```

Deferreds

Finally, a simple demonstration of Deferred support, the unique feature of twisted.web.template . Simply put, any renderer may return a Deferred which fires with some template content instead of the template content itself. As shown above, flattenString will return a Deferred that fires with the full content of the string. But if there's a lot of content, you might not want to wait before starting to send some of it to your HTTP client: for that case, you can use flatten . It's difficult to demonstrate this directly in a browser-based application; unless you insert very long delays before firing your Deferreds, it just looks like your browser is instantly displaying everything. Here's an example that just prints out some HTML template, with markers inserted for where certain events happen:

wait_for_it.py

```
import sys
from twisted.web.template import XMLString, Element, renderer, flatten
from twisted.internet.defer import Deferred

sample = XMLString(
    """
    <div xmlns:t="http://twistedmatrix.com/ns/twisted.web.template/0.1">
    Before waiting ...
    <span t:render="wait"></span>
    ... after waiting.
    </div>
    """)

class WaitForIt(Element):
    def __init__(self):
        Element.__init__(self, loader=sample)
        self.deferred = Deferred()

    @renderer
    def wait(self, request, tag):
        return self.deferred.addCallback(
            lambda aValue: tag("A value: " + repr(aValue)))

def done(ignore):
    print("[[[Deferred fired.]]]")

print('[[[Rendering the template.]]]')
it = WaitForIt()
flatten(None, it, sys.stdout.write).addCallback(done)
print('[[[In progress... now firing the Deferred.]]]')
it.deferred.callback("<value>")
print('[[[All done.]]]')
```

If you run this example, you should get the following output:

waited-for-it.html

```
[[[Rendering the template.]]]
<div>
    Before waiting ...
    [[[In progress... now firing the Deferred.]]]
<span>A value: '&lt;value&gt;'</span>
    ... after waiting.
    </div>[[[Deferred fired.]]]
[[[All done.]]]
```

This demonstrates that part of the output (everything up to "`[[[In progress...` ") is written out immediately

as it's rendered. But once it hits the Deferred, `WaitForIt` 's rendering needs to pause until `.callback(...)` is called on that Deferred. You can see that no further output is produced until the message indicating that the Deferred is being fired is complete. By returning Deferreds and using flatten , you can avoid buffering large amounts of data.

A Brief Note on Formats and DOCTYPEs

The goal of `twisted.web.template` is to emit both valid HTML or XHTML . However, in order to get the maximally standards-compliant output format you desire, you have to know which one you want, and take a few simple steps to emit it correctly. Many browsers will probably work with most output if you ignore this section entirely, but the HTML specification recommends that you specify an appropriate DOCTYPE .

As a `DOCTYPE` declaration in your template would describe the template itself, rather than its output, it won't be included in your output. If you wish to annotate your template output with a DOCTYPE, you will have to write it to the browser out of band. One way to do this would be to simply do `request.write('<!DOCTYPE html>\n')` when you are ready to begin emitting your response. The same goes for an XML `DOCTYPE` declaration.

`twisted.web.template` will remove the `xmlns` attributes used to declare the `http://twistedmatrix.com/ns/twisted.web.template/0.1` namespace, but it will not modify other namespace declaration attributes. Therefore if you wish to serialize in HTML format, you should not use other namespaces; if you wish to serialize to XML, feel free to insert any namespace declarations that are appropriate, and they will appear in your output.

Note: This relaxed approach is correct in many cases. However, in certain contexts - especially <script> and <style> tags - quoting rules differ in significant ways between HTML and XML, and between different browsers' parsers in HTML. If you want to generate dynamic content inside a script or stylesheet, the best option is to load the resource externally so you don't have to worry about quoting rules. The second best option is to strictly configure your content-types and DOCTYPE declarations for XML, whose quoting rules are simple and compatible with the approach that `twisted.web.template` takes. And, please remember: regardless of how you put it there, any user input placed inside a <script> or <style> tag is a potential security issue.

A Bit of History

Those of you who used Divmod Nevow may notice some similarities. `twisted.web.template` is in fact derived from the latest version of Nevow, but includes only the latest components from Nevow's rendering pipeline, and does not have any of the legacy compatibility layers that Nevow grew over time. This should make using `twisted.web.template` a similar experience for many long-time users of Twisted who have previously used Nevow for its twisted-friendly templating, but more straightforward for new users.

Creating XML-RPC Servers and Clients with Twisted

Introduction

XML-RPC is a simple request/reply protocol that runs over HTTP. It is simple, easy to implement and supported by most programming languages. Twisted's XML-RPC support is implemented using the'xmlrpclib <http://docs.python.org/library/xmlrpclib.html>'_ library that is included with Python 2.2 and later.

Creating a XML-RPC server

Making a server is very easy - all you need to do is inherit from twisted.web.xmlrpc.XMLRPC . You then create methods beginning with `xmlrpc_` . The methods' arguments determine what arguments it will accept from XML-RPC clients. The result is what will be returned to the clients.

Methods published via XML-RPC can return all the basic XML-RPC types, such as strings, lists and so on (just return a regular python integer, etc). They can also raise exceptions or return Failure instances to indicate an error has occurred, or `Binary` , `Boolean` or `DateTime` instances (all of these are the same as the respective classes in xmlrpclib. In addition, XML-RPC published methods can return Deferred instances whose results are one of the above. This allows you to return results that can't be calculated immediately, such as database queries. See the *Deferred documentation* for more details.

XMLRPC instances are Resource objects, and they can thus be published using a Site. The following example has two methods published via XML-RPC, `add(a, b)` and `echo(x)` .

```
from twisted.web import xmlrpc, server

class Example(xmlrpc.XMLRPC):
    """
    An example object to be published.
    """

    def xmlrpc_echo(self, x):
        """
        Return all passed args.
        """
        return x

    def xmlrpc_add(self, a, b):
        """
        Return sum of arguments.
        """
        return a + b

    def xmlrpc_fault(self):
        """
        Raise a Fault indicating that the procedure should not be used.
        """
        raise xmlrpc.Fault(123, "The fault procedure is faulty.")

if __name__ == '__main__':
    from twisted.internet import reactor, endpoints
    r = Example()
    endpoint = endpoints.TCP4ServerEndpoint(reactor, 7080)
    endpoint.listen(server.Site(r))
    reactor.run()
```

After we run this command, we can connect with a client and send commands to the server:

```
>>> import xmlrpclib
>>> s = xmlrpclib.Server('http://localhost:7080/')
>>> s.echo("lala")
'lala'
>>> s.add(1, 2)
3
>>> s.fault()
Traceback (most recent call last):
...
xmlrpclib.Fault: <Fault 123: 'The fault procedure is faulty.'>
>>>
```

If the Request object is needed by an `xmlrpc_*` method, it can be made available using the twisted.web.xmlrpc.withRequest decorator. When using this decorator, the method will be passed the request object

as the first argument, before any XML-RPC parameters. For example:

```
from twisted.web.xmlrpc import XMLRPC, withRequest
from twisted.web.server import Site

class Example(XMLRPC):
    @withRequest
    def xmlrpc_headerValue(self, request, headerName):
        return request.requestHeaders.getRawHeaders(headerName)

if __name__ == '__main__':
    from twisted.internet import reactor, endpoints
    endpoint = endpoints.TCP4ServerEndpoint(reactor, 7080)
    endpoint.listen(Site(Example()))
    reactor.run()
```

XML-RPC resources can also be part of a normal Twisted web server, using resource scripts. The following is an example of such a resource script:

`xmlquote.rpy`

```
from twisted.web import xmlrpc
import os

def getQuote():
    return "What are you talking about, William?"

class Quoter(xmlrpc.XMLRPC):

    def xmlrpc_quote(self):
        return getQuote()

resource = Quoter()
```

Using XML-RPC sub-handlers

XML-RPC resource can be nested so that one handler calls another if a method with a given prefix is called. For example, to add support for an XML-RPC method `date.time()` to the `Example` class, you could do the following:

```
import time
from twisted.web import xmlrpc, server

class Example(xmlrpc.XMLRPC):
    """
    An example object to be published.
    """

    def xmlrpc_echo(self, x):
        """
        Return all passed args.
        """
        return x

    def xmlrpc_add(self, a, b):
        """
        Return sum of arguments.
        """
```

```
        return a + b

class Date(xmlrpc.XMLRPC):
    """
    Serve the XML-RPC 'time' method.
    """

    def xmlrpc_time(self):
        """
        Return UNIX time.
        """
        return time.time()

if __name__ == '__main__':
    from twisted.internet import reactor, endpoints
    r = Example()
    date = Date()
    r.putSubHandler('date', date)
    endpoint = endpoints.TCP4ServerEndpoint(reactor, 7080)
    endpoint.listen(server.Site(r))
    reactor.run()
```

By default, a period ('.') separates the prefix from the method name, but you can use a different character by overriding the `XMLRPC.separator` data member in your base XML-RPC server. XML-RPC servers may be nested to arbitrary depths using this method.

Using your own procedure getter

Sometimes, you want to implement your own policy of getting the end implementation. E.g. just like sub-handlers you want to divide the implementations into separate classes but may not want to introduce `XMLRPC.separator` in the procedure name. In such cases just override the `lookupProcedure(self, procedurePath)` method and return the correct callable. Raise twisted.web.xmlrpc.NoSuchFunction otherwise.

`xmlrpc-customized.py`

```
from twisted.web import xmlrpc, server
from twisted.internet import endpoints

class EchoHandler:

    def echo(self, x):
        """
        Return all passed args
        """
        return x



class AddHandler:

    def add(self, a, b):
        """
        Return sum of arguments.
        """
        return a + b
```

```
class Example(xmlrpc.XMLRPC):
    """
    An example of using you own policy to fetch the handler
    """

    def __init__(self):
        xmlrpc.XMLRPC.__init__(self)
        self._addHandler = AddHandler()
        self._echoHandler = EchoHandler()

        #We keep a dict of all relevant
        #procedure names and callable.
        self._procedureToCallable = {
            'add':self._addHandler.add,
            'echo':self._echoHandler.echo
        }

    def lookupProcedure(self, procedurePath):
        try:
            return self._procedureToCallable[procedurePath]
        except KeyError as e:
            raise xmlrpc.NoSuchFunction(self.NOT_FOUND,
                        "procedure %s not found" % procedurePath)

    def listProcedures(self):
        """
        Since we override lookupProcedure, its suggested to override
        listProcedures too.
        """
        return ['add', 'echo']



if __name__ == '__main__':
    from twisted.internet import reactor
    r = Example()
    endpoint = endpoints.TCP4ServerEndpoint(reactor, 7080)
    endpoint.listen(server.Site(r))
    reactor.run()
```

Adding XML-RPC Introspection support

XML-RPC has an informal IntrospectionAPI that specifies three methods in a `system` sub-handler which allow a client to query a server about the server's API. Adding Introspection support to the `Example` class is easy using the XMLRPCIntrospection class:

```
from twisted.web import xmlrpc, server

class Example(xmlrpc.XMLRPC):
    """An example object to be published."""

    def xmlrpc_echo(self, x):
        """Return all passed args."""
        return x
```

```
    xmlrpc_echo.signature = [['string', 'string'],
                             ['int', 'int'],
                             ['double', 'double'],
                             ['array', 'array'],
                             ['struct', 'struct']]

    def xmlrpc_add(self, a, b):
        """Return sum of arguments."""
        return a + b

    xmlrpc_add.signature = [['int', 'int', 'int'],
                            ['double', 'double', 'double']]
    xmlrpc_add.help = "Add the arguments and return the sum."

if __name__ == '__main__':
    from twisted.internet import reactor, endpoints
    r = Example()
    xmlrpc.addIntrospection(r)
    endpoint = endpoints.TCP4ServerEndpoint(reactor, 7080)
    endpoint.listen(server.Site(r))
    reactor.run()
```

Note the method attributes `help` and `signature` which are used by the Introspection API methods `system.methodHelp` and `system.methodSignature` respectively. If no `help` attribute is specified, the method's documentation string is used instead.

SOAP Support

From the point of view of a Twisted developer, there is little difference between XML-RPC support and SOAP support. Here is an example of SOAP usage:

`soap.rpy`

```
from twisted.web import soap
import os

def getQuote():
    return "That beverage, sir, is off the hizzy."

class Quoter(soap.SOAPPublisher):
    """Publish one method, 'quote'."""

    def soap_quote(self):
        return getQuote()

resource = Quoter()
```

Creating an XML-RPC Client

XML-RPC clients in Twisted are meant to look as something which will be familiar either to `xmlrpclib` or to Perspective Broker users, taking features from both, as appropriate. There are two major deviations from the `xmlrpclib` way which should be noted:

1. No implicit `/RPC2` . If the services uses this path for the XML-RPC calls, then it will have to be given explicitly.

2. No magic `__getattr__` : calls must be made by an explicit `callRemote` .

The interface Twisted presents to XML-RPC client is that of a proxy object: twisted.web.xmlrpc.Proxy . The constructor for the object receives a URL: it must be an HTTP or HTTPS URL. When an XML-RPC service is described, the URL to that service will be given there.

Having a proxy object, one can just call the `callRemote` method, which accepts a method name and a variable argument list (but no named arguments, as these are not supported by XML-RPC). It returns a deferred, which will be called back with the result. If there is any error, at any level, the errback will be called. The exception will be the relevant Twisted error in the case of a problem with the underlying connection (for example, a timeout), `IOError` containing the status and message in the case of a non-200 status or a `xmlrpclib.Fault` in the case of an XML-RPC level problem.

```
from twisted.web.xmlrpc import Proxy
from twisted.internet import reactor

def printValue(value):
    print(repr(value))
    reactor.stop()

def printError(error):
    print('error', error)
    reactor.stop()

proxy = Proxy('http://advogato.org/XMLRPC')
proxy.callRemote('test.sumprod', 3, 5).addCallbacks(printValue, printError)
reactor.run()
```

prints:

```
[8, 15]
```

Serving SOAP and XML-RPC simultaneously

twisted.web.xmlrpc.XMLRPC and twisted.web.soap.SOAPPublisher are both Resource s. So, to serve both XML-RPC and SOAP in the one web server, you can use the putChild method of Resource.

The following example uses an empty resource.Resource as the root resource for a Site , and then adds `/RPC2` and `/SOAP` paths to it.

`xmlAndSoapQuote.py`

```
from twisted.web import soap, xmlrpc, resource, server
from twisted.internet import endpoints

import os

def getQuote():
    return "Victory to the burgeois, you capitalist swine!"

class XMLRPCQuoter(xmlrpc.XMLRPC):
    def xmlrpc_quote(self):
        return getQuote()

class SOAPQuoter(soap.SOAPPublisher):
    def soap_quote(self):
        return getQuote()
```

```
def main():
    from twisted.internet import reactor
    root = resource.Resource()
    root.putChild('RPC2', XMLRPCQuoter())
    root.putChild('SOAP', SOAPQuoter())
    endpoint = endpoints.TCP4ServerEndpoint(reactor, 7080)
    endpoint.listen(server.Site(root))
    reactor.run()

if __name__ == '__main__':
    main()
```

Refer to *Twisted Web Development* for more details about Resources.

Twisted Web In 60 Seconds

Serving Static Content From a Directory

The goal of this example is to show you how to serve static content from a filesystem. First, we need to import some objects:

- Site, an IProtocolFactory which glues a listening server port (IListeningPort) to the HTTPChannel implementation:

```
from twisted.web.server import Site
```

- File, an IResource which glues the HTTP protocol implementation to the filesystem:

```
from twisted.web.static import File
```

- The reactor, which drives the whole process, actually accepting TCP connections and moving bytes into and out of them:

```
from twisted.internet import reactor
```

- And the endpoints module, which gives us tools for, amongst other things, creating listening sockets:

```
from twisted.internet import endpoints
```

Next, we create an instance of the File resource pointed at the directory to serve:

```
resource = File("/tmp")
```

Then we create an instance of the Site factory with that resource:

```
factory = Site(resource)
```

Now we glue that factory to a TCP port:

```
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8888)
endpoint.listen(factory)
```

Finally, we start the reactor so it can make the program work:

```
reactor.run()
```

And that's it. Here's the complete program:

```
from twisted.web.server import Site
from twisted.web.static import File
from twisted.internet import reactor, endpoints

resource = File('/tmp')
factory = Site(resource)
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8888)
endpoint.listen(factory)
reactor.run()
```

Bonus example! For those times when you don't actually want to write a new program, the above implemented functionality is one of the things the command line `twistd` tool can do. In this case, the command

```
twistd -n web --path /tmp
```

will accomplish the same thing as the above server. See *helper programs* in the Twisted Core documentation for more information on using `twistd`.

Generating a Page Dynamically

The goal of this example is to show you how to dynamically generate the contents of a page.

Taking care of some of the necessary imports first, we'll import Site, the reactor, and endpoints :

```
from twisted.internet import reactor, endpoints
from twisted.web.server import Site
```

The Site is a factory which associates a listening port with the HTTP protocol implementation. The reactor is the main loop that drives any Twisted application. Endpoints are used to create listening ports.

Next, we'll import one more thing from Twisted Web: Resource . An instance of `Resource` (or a subclass) represents a page (technically, the entity addressed by a URI).

```
from twisted.web.resource import Resource
```

Since we're going to make the demo resource a clock, we'll also import the time module:

```
import time
```

With imports taken care of, the next step is to define a `Resource` subclass which has the dynamic rendering behavior we want. Here's a resource which generates a page giving the time:

```
class ClockPage(Resource):
    isLeaf = True
    def render_GET(self, request):
        return "<html><body>%s</body></html>" % (time.ctime(),)
```

Setting `isLeaf` to `True` indicates that `ClockPage` resources will never have any children.

The `render_GET` method here will be called whenever the URI we hook this resource up to is requested with the `GET` method. The byte string it returns is what will be sent to the browser.

With the resource defined, we can create a `Site` from it:

```
resource = ClockPage()
factory = Site(resource)
```

Just as with the previous static content example, this configuration puts our resource at the very top of the URI hierarchy, ie at `/` . With that `Site` instance, we can tell the reactor to *create a TCP server* and start servicing requests:

```
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8880)
endpoint.listen(factory)
reactor.run()
```

Here's the code with no interruptions:

```
from twisted.internet import reactor, endpoints
from twisted.web.server import Site
from twisted.web.resource import Resource
import time

class ClockPage(Resource):
    isLeaf = True
    def render_GET(self, request):
        return "<html><body>%s</body></html>" % (time.ctime(),)

resource = ClockPage()
factory = Site(resource)
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8880)
endpoint.listen(factory)
reactor.run()
```

Static URL Dispatch

The goal of this example is to show you how to serve different content at different URLs.

The key to understanding how different URLs are handled with the resource APIs in Twisted Web is understanding that any URL can be used to address a node in a tree. Resources in Twisted Web exist in such a tree, and a request for a URL will be responded to by the resource which that URL addresses. The addressing scheme considers only the path segments of the URL. Starting with the root resource (the one used to construct the `Site`) and the first path segment, a child resource is looked up. As long as there are more path segments, this process is repeated using the result of the previous lookup and the next path segment. For example, to handle a request for `"/foo/bar"` , first the root's `"foo"` child is retrieved, then that resource's `"bar"` child is retrieved, then that resource is used to create the response.

With that out of the way, let's consider an example that can serve a few different resources at a few different URLs.

First things first: we need to import Site , the factory for HTTP servers, Resource , a convenient base class for custom pages, reactor , the object which implements the Twisted main loop, and endpoints, which contains classes for creating listening sockets. We'll also import File to use as the resource at one of the example URLs.

```
from twisted.web.server import Site
from twisted.web.resource import Resource
from twisted.internet import reactor, endpoints
from twisted.web.static import File
```

Now we create a resource which will correspond to the root of the URL hierarchy: all URLs are children of this resource.

```
root = Resource()
```

Here comes the interesting part of this example. We're now going to create three more resources and attach them to the three URLs `/foo` , `/bar` , and `/baz` :

```
root.putChild("foo", File("/tmp"))
root.putChild("bar", File("/lost+found"))
root.putChild("baz", File("/opt"))
```

Last, all that's required is to create a `Site` with the root resource, associate it with a listening server port, and start the reactor:

```
factory = Site(root)
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8880)
endpoint.listen(factory)
reactor.run()
```

With this server running, `http://localhost:8880/foo` will serve a listing of files from `/tmp` , `http://localhost:8880/bar` will serve a listing of files from `/lost+found` , and `http://localhost:8880/baz` will serve a listing of files from `/opt` .

Here's the whole example uninterrupted:

```
from twisted.web.server import Site
from twisted.web.resource import Resource
from twisted.internet import reactor, endpoints
from twisted.web.static import File

root = Resource()
root.putChild("foo", File("/tmp"))
root.putChild("bar", File("/lost+found"))
root.putChild("baz", File("/opt"))

factory = Site(root)
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8880)
endpoint.listen(factory)
reactor.run()
```

Dynamic URL Dispatch

In the *previous example* we covered how to statically configure Twisted Web to serve different content at different URLs. The goal of this example is to show you how to do this dynamically instead. Reading the previous installment if you haven't already is suggested in order to get an overview of how URLs are treated when using Twisted Web's resource APIs.

Site (the object which associates a listening server port with the HTTP implementation), Resource (a convenient base class to use when defining custom pages), reactor (the object which implements the Twisted main loop), and endpoints return once again:

```
from twisted.web.server import Site
from twisted.web.resource import Resource
from twisted.internet import reactor, endpoints
```

With that out of the way, here's the interesting part of this example. We're going to define a resource which renders a whole-year calendar. The year it will render the calendar for will be the year in the request URL. So, for example,

`/2009` will render a calendar for 2009. First, here's a resource that renders a calendar for the year passed to its initializer:

```
from calendar import calendar

class YearPage(Resource):
    def __init__(self, year):
        Resource.__init__(self)
        self.year = year

    def render_GET(self, request):
        return "<html><body><pre>%s</pre></body></html>" % (calendar(self.year),)
```

Pretty simple - not all that different from the first dynamic resource demonstrated in *Generating a Page Dynamically* . Now here's the resource that handles URLs with a year in them by creating a suitable instance of this `YearPage` class:

```
class Calendar(Resource):
  def getChild(self, name, request):
      return YearPage(int(name))
```

By implementing getChild here, we've just defined how Twisted Web should find children of `Calendar` instances when it's resolving an URL into a resource. This implementation defines all integers as the children of `Calendar` (and punts on error handling, more on that later).

All that's left is to create a `Site` using this resource as its root and then start the reactor:

```
root = Calendar()
factory = Site(root)
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8880)
endpoint.listen(factory)
reactor.run()
```

And that's all. Any resource-based dynamic URL handling is going to look basically like `Calendar.getChild` . Here's the full example code:

```
from twisted.web.server import Site
from twisted.web.resource import Resource
from twisted.internet import reactor, endpoints

from calendar import calendar

class YearPage(Resource):
    def __init__(self, year):
        Resource.__init__(self)
        self.year = year

    def render_GET(self, request):
        return "<html><body><pre>%s</pre></body></html>" % (calendar(self.year),)

class Calendar(Resource):
  def getChild(self, name, request):
      return YearPage(int(name))

root = Calendar()
factory = Site(root)
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8880)
endpoint.listen(factory)
```

```
reactor.run()
```

Error Handling

In this example we'll extend dynamic dispatch to return a 404 (not found) response when a client requests a non-existent URL.

As in the previous examples, we'll start with Site , Resource , reactor, and endpoints imports:

```
from twisted.web.server import Site
from twisted.web.resource import Resource
from twisted.internet import reactor, endpoints
```

Next, we'll add one more import. NoResource is one of the pre-defined error resources provided by Twisted Web. It generates the necessary 404 response code and renders a simple html page telling the client there is no such resource.

```
from twisted.web.resource import NoResource
```

Next, we'll define a custom resource which does some dynamic URL dispatch. This example is going to be just like the *previous one* , where the path segment is interpreted as a year; the difference is that this time we'll handle requests which don't conform to that pattern by returning the not found response:

```
class Calendar(Resource):
    def getChild(self, name, request):
        try:
            year = int(name)
        except ValueError:
            return NoResource()
        else:
            return YearPage(year)
```

Aside from including the definition of `YearPage` from the previous example, the only other thing left to do is the normal `Site` and `reactor` setup. Here's the complete code for this example:

```
from twisted.web.server import Site
from twisted.web.resource import Resource
from twisted.internet import reactor, endpoints
from twisted.web.resource import NoResource

from calendar import calendar

class YearPage(Resource):
    def __init__(self, year):
        Resource.__init__(self)
        self.year = year

    def render_GET(self, request):
        return "<html><body><pre>%s</pre></body></html>" % (calendar(self.year),)

class Calendar(Resource):
    def getChild(self, name, request):
        try:
            year = int(name)
        except ValueError:
            return NoResource()
        else:
```

```
            return YearPage(year)

root = Calendar()
factory = Site(root)
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8880)
endpoint.listen(factory)
reactor.run()
```

This server hands out the same calendar views as the one from the previous installment, but it will also hand out a nice error page with a 404 response when a request is made for a URL which cannot be interpreted as a year.

Custom Response Codes

The previous example introduced NoResource , a Twisted Web error resource which responds with a 404 (not found) code. This example will cover the APIs that `NoResource` uses to do this so that you can generate your own custom response codes as desired.

First, the now-standard import preamble:

```
from twisted.web.server import Site
from twisted.web.resource import Resource
from twisted.internet import reactor, endpoints
```

Now we'll define a new resource class that always returns a 402 (payment required) response. This is really not very different from the resources that was defined in previous examples. The fact that it has a response code other than 200 doesn't change anything else about its role. This will require using the request object, though, which none of the previous examples have done.

The Request object has shown up in a couple of places, but so far we've ignored it. It is a parameter to the getChild API as well as to render methods such as `render_GET` . As you might have suspected, it represents the request for which a response is to be generated. Additionally, it also represents the response being generated. In this example we're going to use its setResponseCode method to - you guessed it - set the response's status code.

```
class PaymentRequired(Resource):
    def render_GET(self, request):
        request.setResponseCode(402)
        return "<html><body>Please swipe your credit card.</body></html>"
```

Just like the other resources I've demonstrated, this one returns a string from its `render_GET` method to define the body of the response. All that's different is the call to `setResponseCode` to override the default response code, 200, with a different one.

Finally, the code to set up the site and reactor. We'll put an instance of the above defined resource at `/buy` :

```
root = Resource()
root.putChild("buy", PaymentRequired())
factory = Site(root)
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8880)
endpoint.listen(factory)
reactor.run()
```

Here's the complete example:

```
from twisted.web.server import Site
from twisted.web.resource import Resource
from twisted.internet import reactor, endpoints
```

```
class PaymentRequired(Resource):
    def render_GET(self, request):
        request.setResponseCode(402)
        return "<html><body>Please swipe your credit card.</body></html>"

root = Resource()
root.putChild("buy", PaymentRequired())
factory = Site(root)
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8880)
endpoint.listen(factory)
reactor.run()
```

Run the server and visit `http://localhost:8880/buy` in your browser. It'll look pretty boring, but if you use Firefox's View Page Info right-click menu item (or your browser's equivalent), you'll be able to see that the server indeed sent back a 402 response code.

Handling POSTs

All of the previous examples have focused on `GET` requests. Unlike `GET` requests, `POST` requests can have a request body - extra data after the request headers; for example, data representing the contents of an HTML form. Twisted Web makes this data available to applications via the Request object.

Here's an example web server which renders a static HTML form and then generates a dynamic page when that form is posted back to it. Disclaimer: While it's convenient for this example, it's often not a good idea to make a resource that `POST` s to itself; this isn't about Twisted Web, but the nature of HTTP in general; if you do this in a real application, make sure you understand the possible negative consequences.

As usual, we start with some imports. In addition to the Twisted imports, this example uses the `cgi` module to escape user-enteredcontent for inclusion in the output.

```
from twisted.web.server import Site
from twisted.web.resource import Resource
from twisted.internet import reactor, endpoints

import cgi
```

Next, we'll define a resource which is going to do two things. First, it will respond to `GET` requests with a static HTML form:

```
class FormPage(Resource):
    def render_GET(self, request):
        return '<html><body><form method="POST"><input name="the-field" type="text" />
↪</form></body></html>'
```

This is similar to the resource used in a *previous installment* . However, we'll now add one more method to give it a second behavior; this `render_POST` method will allow it to accept `POST` requests:

```
...
    def render_POST(self, request):
        return '<html><body>You submitted: %s</body></html>' % (cgi.escape(request.
↪args["the-field"][0]),)
```

The main thing to note here is the use of `request.args` . This is a dictionary-like object that provides access to the contents of the form. The keys in this dictionary are the names of inputs in the form. Each value is a list containing strings (since there can be multiple inputs with the same name), which is why we had to extract the first element to pass to `cgi.escape` . `request.args` will be populated from form contents whenever a `POST` request is made with a

content type of `application/x-www-form-urlencoded` or `multipart/form-data` (it's also populated by query arguments for any type of request).

Finally, the example just needs the usual site creation and port setup:

```
root = Resource()
root.putChild("form", FormPage())
factory = Site(root)
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8880)
endpoint.listen(factory)
reactor.run()
```

Run the server and visit http://localhost:8880/form , submit the form, and watch it generate a page including the value you entered into the single field.

Here's the complete source for the example:

```
from twisted.web.server import Site
from twisted.web.resource import Resource
from twisted.internet import reactor, endpoints

import cgi

class FormPage(Resource):
    def render_GET(self, request):
        return '<html><body><form method="POST"><input name="the-field" type="text" />
↪</form></body></html>'

    def render_POST(self, request):
        return '<html><body>You submitted: %s</body></html>' % (cgi.escape(request.
↪args["the-field"][0]),)

root = Resource()
root.putChild("form", FormPage())
factory = Site(root)
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8880)
endpoint.listen(factory)
reactor.run()
```

Other Request Bodies

The previous example demonstrated how to accept the payload of a `POST` carrying HTML form data. What about `POST` requests with data in some other format? Or even `PUT` requests? Here is an example which demonstrates how to get *any* request body, regardless of its format - using the request's content attribute.

The only significant difference between this example and the previous is that instead of accessing `request.args` in `render_POST` , it uses `request.content` to get the request's body directly:

```
...
    def render_POST(self, request):
        return '<html><body>You submitted: %s</body></html>' % (cgi.escape(request.
↪content.read()),)
```

`request.content` is a file-like object, so the body is read from it. The exact type may vary, so avoid relying on non-file methods you may find (such as `getvalue` when happens to be a `StringIO` instance).

Here's the complete source for this example - again, almost identical to the previous `POST` example, with only `render_POST` changed:

```
from twisted.web.server import Site
from twisted.web.resource import Resource
from twisted.internet import reactor, endpoints

import cgi

class FormPage(Resource):
    def render_GET(self, request):
        return '<html><body><form method="POST"><input name="the-field" type="text" />
↪</form></body></html>'

    def render_POST(self, request):
        return '<html><body>You submitted: %s</body></html>' % (cgi.escape(request.
↪content.read()),)

root = Resource()
root.putChild("form", FormPage())
factory = Site(root)
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8880)
endpoint.listen(factory)
reactor.run()
```

rpy scripts (or, how to save yourself some typing)

The goal of this installment is to show you another way to run a Twisted Web server with a custom resource which doesn't require as much code as the previous examples.

The feature in question is called an `rpy script`. An rpy script is a Python source file which defines a resource and can be loaded into a Twisted Web server. The advantages of this approach are that you don't have to write code to create the site or set up a listening port with the reactor. That means fewer lines of code that aren't dedicated to the task you're trying to accomplish.

There are some disadvantages, though. An rpy script must have the extension `.rpy`. This means you can't import it using the usual Python import statement. This means it's hard to re-use code in an rpy script. This also means you can't easily unit test it. The code in an rpy script is evaluated in an unusual context. So, while rpy scripts may be useful for testing out ideas, they're not recommend for much more than that.

Okay, with that warning out of the way, let's dive in. First, as mentioned, rpy scripts are Python source files with the `.rpy` extension. So, open up an appropriately named file (for example, `example.rpy`) and put this code in it:

```
import time

from twisted.web.resource import Resource

class ClockPage(Resource):
    isLeaf = True
    def render_GET(self, request):
        return "<html><body>%s</body></html>" % (time.ctime(),)

resource = ClockPage()
```

You may recognize this as the resource from the *first dynamic rendering example*. What's different is what you don't see: we didn't import `reactor` or `Site`. There are no calls to `endpoints.TCP4ServerEndpoint` or `run`. Instead, and this is the core idea for rpy scripts, we just bound the name `resource` to the resource we want the script to serve. Every rpy script must bind this name, and this name is the only thing Twisted Web will pay attention to in an rpy script.

All that's left is to drop this rpy script into a Twisted Web server. There are a few ways to do this. The simplest way is with `twistd`:

```
$ twistd -n web --path .
```

Hit http://localhost:8080/example.rpy to see it run. You can pass other arguments here too. `twistd web` has options for specifying which port number to bind, whether to set up an HTTPS server, and plenty more. Other options you can pass to `twistd` allow you to configure logging to work differently, to select a different reactor, etc. For a full list of options, see `twistd --help` and `twistd web --help`.

Asynchronous Responses

In all of the previous examples, the resource examples presented generated responses immediately. One of the features of prime interest of Twisted Web, though, is the ability to generate a response over a longer period of time while leaving the server free to respond to other requests. In other words, asynchronously. In this installment, we'll write a resource like this.

A resource that generates a response asynchronously looks like one that generates a response synchronously in many ways. The same base class, Resource, is used either way; the same render methods are used. There are three basic differences, though.

First, instead of returning the string which will be used as the body of the response, the resource uses Request.write. This method can be called repeatedly. Each call appends another string to the response body. Second, when the entire response body has been passed to `Request.write`, the application must call Request.finish. As you might expect from the name, this ends the response. Finally, in order to make Twisted Web not end the response as soon as the render method returns, the render method must return `NOT_DONE_YET`. Consider this example:

```
from twisted.web.resource import Resource
from twisted.web.server import NOT_DONE_YET
from twisted.internet import reactor

class DelayedResource(Resource):
    def _delayedRender(self, request):
        request.write("<html><body>Sorry to keep you waiting.</body></html>")
        request.finish()

    def render_GET(self, request):
        reactor.callLater(5, self._delayedRender, request)
        return NOT_DONE_YET
```

If you're not familiar with the reactor callLater method, all you really need to know about it to understand this example is that the above usage of it arranges to have `self._delayedRender(request)` run about 5 seconds after `callLater` is invoked from this render method and that it returns immediately.

All three of the elements mentioned earlier can be seen in this example. The resource uses `Request.write` to set the response body. It uses `Request.finish` after the entire body has been specified (all with just one call to write in this case). Lastly, it returns `NOT_DONE_YET` from its render method. So there you have it, asynchronous rendering with Twisted Web.

Here's a complete rpy script based on this resource class (see the *previous example* if you need a reminder about rpy scripts):

```
from twisted.web.resource import Resource
from twisted.web.server import NOT_DONE_YET
from twisted.internet import reactor

class DelayedResource(Resource):
```

```
    def _delayedRender(self, request):
        request.write("<html><body>Sorry to keep you waiting.</body></html>")
        request.finish()

    def render_GET(self, request):
        reactor.callLater(5, self._delayedRender, request)
        return NOT_DONE_YET

resource = DelayedResource()
```

Drop this source into a `.rpy` file and fire up a server using `twistd -n web --path /directory/containing/script/`. You'll see that loading the page takes 5 seconds. If you try to load a second before the first completes, it will also take 5 seconds from the time you request it (but it won't be delayed by any other outstanding requests).

Something else to consider when generating responses asynchronously is that the client may not wait around to get the response to its request. A *subsequent example* demonstrates how to detect that the client has abandoned the request and that the server shouldn't bother to finish generating its response.

Asynchronous Responses (via Deferred)

The previous example had a Resource that generates its response asynchronously rather than immediately upon the call to its render method. Though it was a useful demonstration of the `NOT_DONE_YET` feature of Twisted Web, the example didn't reflect what a realistic application might want to do. This example introduces Deferred , the Twisted class which is used to provide a uniform interface to many asynchronous events, and shows you an example of using a `Deferred` -returning API to generate an asynchronous response to a request in Twisted Web.

`Deferred` is the result of two consequences of the asynchronous programming approach. First, asynchronous code is frequently (if not always) concerned with some data (in Python, an object) which is not yet available but which probably will be soon. Asynchronous code needs a way to define what will be done to the object once it does exist. It also needs a way to define how to handle errors in the creation or acquisition of that object. These two needs are satisfied by the *callbacks* and *errbacks* of a `Deferred` . Callbacks are added to a `Deferred` with Deferred.addCallback ; errbacks are added with Deferred.addErrback . When the object finally does exist, it is passed to Deferred.callback which passes it on to the callback added with `addCallback` . Similarly, if an error occurs, Deferred.errback is called and the error is passed along to the errback added with `addErrback` . Second, the events that make asynchronous code actually work often take many different, incompatible forms. `Deferred` acts as the uniform interface which lets different parts of an asynchronous application interact and isolates them from implementation details they shouldn't be concerned with.

That's almost all there is to `Deferred` . To solidify your new understanding, now consider this rewritten version of `DelayedResource` which uses a `Deferred` -based delay API. It does exactly the same thing as the *previous example* . Only the implementation is different.

First, the example must import that new API that was just mentioned, deferLater :

```
from twisted.internet.task import deferLater
```

Next, all the other imports (these are the same as last time):

```
from twisted.web.resource import Resource
from twisted.web.server import NOT_DONE_YET
from twisted.internet import reactor
```

With the imports done, here's the first part of the `DelayedResource` implementation. Again, this part of the code is identical to the previous version:

```
class DelayedResource(Resource):
    def _delayedRender(self, request):
        request.write("<html><body>Sorry to keep you waiting.</body></html>")
        request.finish()
```

Next we need to define the render method. Here's where things change a bit. Instead of using callLater , We're going to use deferLater this time. `deferLater` accepts a reactor, delay (in seconds, as with `callLater`), and a function to call after the delay to produce that elusive object discussed in the description of `Deferred` s. We're also going to use `_delayedRender` as the callback to add to the `Deferred` returned by `deferLater` . Since it expects the request object as an argument, we're going to set up the `deferLater` call to return a `Deferred` which has the request object as its result.

```
...
    def render_GET(self, request):
        d = deferLater(reactor, 5, lambda: request)
```

The `Deferred` referenced by `d` now needs to have the `_delayedRender` callback added to it. Once this is done, `_delayedRender` will be called with the result of `d` (which will be `request` , of course — the result of `(lambda: request)()`).

```
...
        d.addCallback(self._delayedRender)
```

Finally, the render method still needs to return `NOT_DONE_YET` , for exactly the same reasons as it did in the previous version of the example.

```
...
        return NOT_DONE_YET
```

And with that, `DelayedResource` is now implemented based on a `Deferred` . The example still isn't very realistic, but remember that since `Deferred` s offer a uniform interface to many different asynchronous event sources, this code now resembles a real application even more closely; you could easily replace `deferLater` with another `Deferred` -returning API and suddenly you might have a resource that does something useful.

Finally, here's the complete, uninterrupted example source, as an rpy script:

```
from twisted.internet.task import deferLater
from twisted.web.resource import Resource
from twisted.web.server import NOT_DONE_YET
from twisted.internet import reactor

class DelayedResource(Resource):
    def _delayedRender(self, request):
        request.write("<html><body>Sorry to keep you waiting.</body></html>")
        request.finish()

    def render_GET(self, request):
        d = deferLater(reactor, 5, lambda: request)
        d.addCallback(self._delayedRender)
        return NOT_DONE_YET

resource = DelayedResource()
```

Interrupted Responses

The previous example had a Resource that generates its response asynchronously rather than immediately upon the call to its render method. When generating responses asynchronously, the possibility is introduced that the connection to the client may be lost before the response is generated. In such a case, it is often desirable to abandon the response generation entirely, since there is nothing to do with the data once it is produced. This example shows how to be notified that the connection has been lost.

This example will build upon the *asynchronous responses example* which simply (if not very realistically) generated its response after a fixed delay. We will expand that resource so that as soon as the client connection is lost, the delayed event is cancelled and the response is never generated.

The feature this example relies on is provided by another Request method: notifyFinish . This method returns a new Deferred which will fire with `None` if the request is successfully responded to or with an error otherwise - for example if the connection is lost before the response is sent.

The example starts in a familiar way, with the requisite Twisted imports and a resource class with the same `_delayedRender` used previously:

```
from twisted.web.resource import Resource
from twisted.web.server import NOT_DONE_YET
from twisted.internet import reactor

class DelayedResource(Resource):
    def _delayedRender(self, request):
        request.write("<html><body>Sorry to keep you waiting.</body></html>")
        request.finish()
```

Before defining the render method, we're going to define an errback (an errback being a callback that gets called when there's an error), though. This will be the errback attached to the `Deferred` returned by `Request.notifyFinish` . It will cancel the delayed call to `_delayedRender` .

```
...
    def _responseFailed(self, err, call):
        call.cancel()
```

Finally, the render method will set up the delayed call just as it did before, and return `NOT_DONE_YET` likewise. However, it will also use `Request.notifyFinish` to make sure `_responseFailed` is called if appropriate.

```
...
    def render_GET(self, request):
        call = reactor.callLater(5, self._delayedRender, request)
        request.notifyFinish().addErrback(self._responseFailed, call)
        return NOT_DONE_YET
```

Notice that since `_responseFailed` needs a reference to the delayed call object in order to cancel it, we passed that object to `addErrback` . Any additional arguments passed to `addErrback` (or `addCallback`) will be passed along to the errback after the Failure instance which is always passed as the first argument. Passing `call` here means it will be passed to `_responseFailed` , where it is expected and required.

That covers almost all the code for this example. Here's the entire example without interruptions, as an *rpy script* :

```
from twisted.web.resource import Resource
from twisted.web.server import NOT_DONE_YET
from twisted.internet import reactor

class DelayedResource(Resource):
    def _delayedRender(self, request):
```

```
        request.write("<html><body>Sorry to keep you waiting.</body></html>")
        request.finish()

    def _responseFailed(self, err, call):
        call.cancel()

    def render_GET(self, request):
        call = reactor.callLater(5, self._delayedRender, request)
        request.notifyFinish().addErrback(self._responseFailed, call)
        return NOT_DONE_YET

resource = DelayedResource()
```

Toss this into `example.rpy`, fire it up with `twistd -n web --path .`, and hit http://localhost:8080/example.rpy . If you wait five seconds, you'll get the page content. If you interrupt the request before then, say by hitting escape (in Firefox, at least), then you'll see perhaps the most boring demonstration ever - no page content, and nothing in the server logs. Success!

Logging Errors

The *previous example* created a server that dealt with response errors by aborting response generation, potentially avoiding pointless work. However, it did this silently for any error. In this example, we'll modify the previous example so that it logs each failed response.

This example will use the Twisted API for logging errors. As was mentioned in the *first example covering Deferreds*, errbacks are passed an error. In the previous example, the `_responseFailed` errback accepted this error as a parameter but ignored it. The only way this example will differ is that this `_responseFailed` will use that error parameter to log a message.

This example will require all of the imports required by the previous example plus one new import:

```
from twisted.python.log import err
```

The only other part of the previous example which changes is the `_responseFailed` callback, which will now log the error passed to it:

```
...
    def _responseFailed(self, failure, call):
        call.cancel()
        err(failure, "Async response demo interrupted response")
```

We're passing two arguments to err here. The first is the error which is being passed in to the callback. This is always an object of type Failure , a class which represents an exception and (sometimes, but not always) a traceback. `err` will format this nicely for the log. The second argument is a descriptive string that tells someone reading the log what the source of the error was.

Here's the full example with the two above modifications:

```
from twisted.web.resource import Resource
from twisted.web.server import NOT_DONE_YET
from twisted.internet import reactor
from twisted.python.log import err

class DelayedResource(Resource):
    def _delayedRender(self, request):
        request.write("<html><body>Sorry to keep you waiting.</body></html>")
        request.finish()
```

```
    def _responseFailed(self, failure, call):
        call.cancel()
        err(failure, "Async response demo interrupted response")

    def render_GET(self, request):
        call = reactor.callLater(5, self._delayedRender, request)
        request.notifyFinish().addErrback(self._responseFailed, call)
        return NOT_DONE_YET

resource = DelayedResource()
```

Run this server as in the *previous example* and interrupt a request. Unlike the previous example, where the server gave no indication that this had happened, you'll see a message in the log output with this version.

Access Logging

As long as we're on the topic of *logging*, this is probably a good time to mention Twisted Web's access log support. In this example, we'll see what Twisted Web logs for each request it processes and how this can be customized.

If you've run any of the previous examples and watched the output of `twistd` or read `twistd.log` then you've already seen some log lines like this:

> 2014-01-29 17:50:50-0500 [HTTPChannel,0,127.0.0.1] "127.0.0.1" - - [29/Jan/2014:22:50:50 +0000] "GET / HTTP/1.1" 200 2753 "-" "Mozilla/5.0 ..."

If you focus on the latter portion of this log message you'll see something that looks like a standard "combined log format" message. However, it's prefixed with the normal Twisted logging prefix giving a timestamp and some protocol and peer addressing information. Much of this information is redundant since it is part of the combined log format. Site lets you produce a more compact log which omits the normal Twisted logging prefix. To take advantage of this feature all that is necessary is to tell Site where to write this compact log. Do this by passing `logPath` to the initializer:

```
...
factory = Site(root, logPath=b"/tmp/access-logging-demo.log")
```

Or if you want to change the logging behavior of a server you're launching with `twistd web` then just pass the `--logfile` option:

```
$ twistd -n web --logfile /tmp/access-logging-demo.log
```

Apart from this, the rest of the server setup is the same. Once you pass `logPath` or use `--logfile` on the command line the server will produce a log file containing lines like:

> "127.0.0.1" - - [30/Jan/2014:00:13:35 +0000] "GET / HTTP/1.1" 200 2753 "-" "Mozilla/5.0 ..."

Any tools expecting combined log format messages should be able to work with these log files.

Site also allows the log format used to be customized using its `logFormatter` argument. Twisted Web comes with one alternate formatter, proxiedLogFormatter, which is for use behind a proxy that sets the `X-Forwarded-For` header. It logs the client address taken from this header rather than the network address of the client directly connected to the server. Here's the complete code for an example that uses both these features:

```
from twisted.web.http import proxiedLogFormatter
from twisted.web.server import Site
from twisted.web.static import File
from twisted.internet import reactor, endpoints

resource = File('/tmp')
```

```
factory = Site(resource, logPath=b"/tmp/access-logging-demo.log",␣
↪logFormatter=proxiedLogFormatter)
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8888)
endpoint.listen(factory)
reactor.run()
```

WSGI

The goal of this example is to show you how to use WSGIResource , another existing Resource subclass, to serve WSGI applications in a Twisted Web server.

Note that `WSGIResource` is a multithreaded WSGI container. Like any other WSGI container, you can't do anything asynchronous in your WSGI applications, even though this is a Twisted WSGI container.

The first new thing in this example is the import of `WSGIResource` :

```
from twisted.web.wsgi import WSGIResource
```

Nothing too surprising there. We still need one of the other usual suspects, too:

```
from twisted.internet import reactor
```

You'll see why in a minute. Next, we need a WSGI application. Here's a really simple one just to get things going:

```
def application(environ, start_response):
    start_response('200 OK', [('Content-type', 'text/plain')])
    return ['Hello, world!']
```

If this doesn't make sense to you, take a look at one of these fine tutorials . Otherwise, or once you're done with that, the next step is to create a `WSGIResource` instance, as this is going to be another *rpy script* example:

```
resource = WSGIResource(reactor, reactor.getThreadPool(), application)
```

Let's dwell on this line for a minute. The first parameter passed to `WSGIResource` is the reactor. Despite the fact that the reactor is global and any code that wants it can always just import it (as, in fact, this rpy script simply does itself), passing it around as a parameter leaves the door open for certain future possibilities - for example, having more than one reactor. There are also testing implications. Consider how much easier it is to unit test a function that accepts a reactor - perhaps a mock reactor specially constructed to make your tests easy to write - rather than importing the real global reactor. That's why `WSGIResource` requires you to pass the reactor to it.

The second parameter passed to `WSGIResource` is a ThreadPool . `WSGIResource` uses this to actually call the application object passed in to it. To keep this example short, we're passing in the reactor's internal threadpool here, letting us skip its creation and shutdown-time destruction. For finer control over how many WSGI requests are served in parallel, you may want to create your own thread pool to use with your `WSGIResource` , but for simple testing, using the reactor's is fine.

The final argument is the application object. This is pretty typical of how WSGI containers work.

The example, sans interruption:

```
from twisted.web.wsgi import WSGIResource
from twisted.internet import reactor

def application(environ, start_response):
    start_response('200 OK', [('Content-type', 'text/plain')])
    return ['Hello, world!']
```

```
resource = WSGIResource(reactor, reactor.getThreadPool(), application)
```

Up to the point where the `WSGIResource` instance defined here exists in the resource hierarchy, the normal resource traversal rules apply: getChild will be called to handle each segment. Once the `WSGIResource` is encountered, though, that process stops and all further URL handling is the responsibility of the WSGI application. This application does nothing with the URL, though, so you won't be able to tell that.

Oh, and as was the case with the first static file example, there's also a command line option you can use to avoid a lot of this. If you just put the above application function, without all of the `WSGIResource` stuff, into a file, say, `foo.py` , then you can launch a roughly equivalent server like this:

```
$ twistd -n web --wsgi foo.application
```

HTTP Authentication

Many of the previous examples have looked at how to serve content by using existing resource classes or implementing new ones. In this example we'll use Twisted Web's basic or digest HTTP authentication to control access to these resources.

guard , the Twisted Web module which provides most of the APIs that will be used in this example, helps you to add authentication and authorization to a resource hierarchy. It does this by providing a resource which implements getChild to return a *dynamically selected resource* . The selection is based on the authentication headers in the request. If those headers indicate that the request is made on behalf of Alice, then Alice's resource will be returned. If they indicate that it was made on behalf of Bob, his will be returned. If the headers contain invalid credentials, an error resource is returned. Whatever happens, once this resource is returned, URL traversal continues as normal from that resource.

The resource that implements this is HTTPAuthSessionWrapper , though it is directly responsible for very little of the process. It will extract headers from the request and hand them off to a credentials factory to parse them according to the appropriate standards (eg HTTPAuthentication: Basic and Digest Access Authentication) and then hand the resulting credentials object off to a Portal , the core of *Twisted Cred* , a system for uniform handling of authentication and authorization. We won't discuss Twisted Cred in much depth here. To make use of it with Twisted Web, the only thing you really need to know is how to implement an IRealm .

You need to implement a realm because the realm is the object that actually decides which resources are used for which users. This can be as complex or as simple as it suitable for your application. For this example we'll keep it very simple: each user will have a resource which is a static file listing of the `public_html` directory in their UNIX home directory. First, we need to import `implements` from `zope.interface` and `IRealm` from `twisted.cred.portal` . Together these will let me mark this class as a realm (this is mostly - but not entirely - a documentation thing). We'll also need File for the actual implementation later.

```
from zope.interface import implementer

from twisted.cred.portal import IRealm
from twisted.web.static import File

@implementer(IRealm)
class PublicHTMLRealm(object):
```

A realm only needs to implement one method: requestAvatar . This method is called after any successful authentication attempt (ie, Alice supplied the right password). Its job is to return the *avatar* for the user who succeeded in authenticating. An *avatar* is just an object that represents a user. In this case, it will be a `File` . In general, with `Guard` , the avatar must be a resource of some sort.

```
...
    def requestAvatar(self, avatarId, mind, *interfaces):
        if IResource in interfaces:
            return (IResource, File("/home/%s/public_html" % (avatarId,)), lambda:␣
↪None)
        raise NotImplementedError()
```

A few notes on this method:

- The `avatarId` parameter is essentially the username. It's the job of some other code to extract the username from the request headers and make sure it gets passed here.
- The `mind` is always `None` when writing a realm to be used with `Guard` . You can ignore it until you want to write a realm for something else.
- `Guard` is always passed `IResource` as the `interfaces` parameter. If `interfaces` only contains interfaces your code doesn't understand, raising `NotImplementedError` is the thing to do, as above. You'll only need to worry about getting a different interface when you write a realm for something other than `Guard` .
- If you want to track when a user logs out, that's what the last element of the returned tuple is for. It will be called when this avatar logs out. `lambda: None` is the idiomatic no-op logout function.
- Notice that the path handling code in this example is written very poorly. This example may be vulnerable to certain unintentional information disclosure attacks. This sort of problem is exactly the reason FilePath exists. However, that's an example for another day...

We're almost ready to set up the resource for this example. To create an `HTTPAuthSessionWrapper` , though, we need two things. First, a portal, which requires the realm above, plus at least one credentials checker:

```
from twisted.cred.portal import Portal
from twisted.cred.checkers import FilePasswordDB

portal = Portal(PublicHTMLRealm(), [FilePasswordDB('httpd.password')])
```

FilePasswordDB is the credentials checker. It knows how to read `passwd(5)` -style (loosely) files to check credentials against. It is responsible for the authentication work after `HTTPAuthSessionWrapper` extracts the credentials from the request.

Next we need either BasicCredentialFactory or DigestCredentialFactory . The former knows how to challenge HTTP clients to do basic authentication; the latter, digest authentication. We'll use digest here:

```
from twisted.web.guard import DigestCredentialFactory

credentialFactory = DigestCredentialFactory("md5", "example.org")
```

The two parameters to this constructor are the hash algorithm and the HTTP authentication realm which will be used. The only other valid hash algorithm is "sha" (but be careful, MD5 is more widely supported than SHA). The HTTP authentication realm is mostly just a string that is presented to the user to let them know why they're authenticating (you can read more about this in the RFC).

With those things created, we can finally instantiate `HTTPAuthSessionWrapper` :

```
from twisted.web.guard import HTTPAuthSessionWrapper

resource = HTTPAuthSessionWrapper(portal, [credentialFactory])
```

There's just one last thing that needs to be done here. When *rpy scripts* were introduced, it was mentioned that they are evaluated in an unusual context. This is the first example that actually needs to take this into account. It so happens that `DigestCredentialFactory` instances are stateful. Authentication will only succeed if the same instance is used

to both generate challenges and examine the responses to those challenges. However, the normal mode of operation for an rpy script is for it to be re-executed for every request. This leads to a new `DigestCredentialFactory` being created for every request, preventing any authentication attempt from ever succeeding.

There are two ways to deal with this. First, and the better of the two ways, we could move almost all of the code into a real Python module, including the code that instantiates the `DigestCredentialFactory`. This would ensure that the same instance was used for every request. Second, and the easier of the two ways, we could add a call to `cache()` to the beginning of the rpy script:

```
cache()
```

`cache` is part of the globals of any rpy script, so you don't need to import it (it's okay to be cringing at this point). Calling `cache` makes Twisted re-use the result of the first evaluation of the rpy script for subsequent requests too - just what we want in this case.

Here's the complete example (with imports re-arranged to the more conventional style):

```
cache()

from zope.interface import implementer

from twisted.cred.portal import IRealm, Portal
from twisted.cred.checkers import FilePasswordDB
from twisted.web.static import File
from twisted.web.resource import IResource
from twisted.web.guard import HTTPAuthSessionWrapper, DigestCredentialFactory

@implementer(IRealm)
class PublicHTMLRealm(object):
    def requestAvatar(self, avatarId, mind, *interfaces):
        if IResource in interfaces:
            return (IResource, File("/home/%s/public_html" % (avatarId,)), lambda:␣
↪None)
        raise NotImplementedError()

portal = Portal(PublicHTMLRealm(), [FilePasswordDB('httpd.password')])

credentialFactory = DigestCredentialFactory("md5", "localhost:8080")
resource = HTTPAuthSessionWrapper(portal, [credentialFactory])
```

And voila, a password-protected per-user Twisted Web server.

Session Basics

Sessions are the most complicated topic covered in this series of examples, and because of that it is going to take a few examples to cover all of the different aspects. This first example demonstrates the very basics of the Twisted Web session API: how to get the session object for the current request and how to prematurely expire a session.

Before diving into the APIs, let's look at the big picture of sessions in Twisted Web. Sessions are represented by instances of Session . The Site creates a new instance of `Session` the first time an application asks for it for a particular session. `Session` instances are kept on the `Site` instance until they expire (due to inactivity or because they are explicitly expired). Each time after the first that a particular session's `Session` object is requested, it is retrieved from the `Site` .

With the conceptual underpinnings of the upcoming API in place, here comes the example. This will be a very simple *rpy script* which tells a user what its unique session identifier is and lets it prematurely expire the session.

First, we'll import Resource so we can define a couple of subclasses of it:

```
from twisted.web.resource import Resource
```

Next we'll define the resource which tells the client what its session identifier is. This is done easily by first getting the session object using Request.getSession and then getting the session object's uid attribute:

```
class ShowSession(Resource):
    def render_GET(self, request):
        return 'Your session id is: ' + request.getSession().uid
```

To let the client expire its own session before it times out, we'll define another resource which expires whatever session it is requested with. This is done using the Session.expire method:

```
class ExpireSession(Resource):
    def render_GET(self, request):
        request.getSession().expire()
        return 'Your session has been expired.'
```

Finally, to make the example an rpy script, we'll make an instance of `ShowSession` and give it an instance of `ExpireSession` as a child using Resource.putChild :

```
resource = ShowSession()
resource.putChild("expire", ExpireSession())
```

And that is the complete example. You can fire this up and load the top page. You'll see a (rather opaque) session identifier that remains the same across reloads (at least until you flush the `TWISTED_SESSION` cookie from your browser or enough time passes). You can then visit the `expire` child and go back to the top page and see that you have a new session.

Here's the complete source for the example:

```
from twisted.web.resource import Resource

class ShowSession(Resource):
    def render_GET(self, request):
        return 'Your session id is: ' + request.getSession().uid

class ExpireSession(Resource):
    def render_GET(self, request):
        request.getSession().expire()
        return 'Your session has been expired.'

resource = ShowSession()
resource.putChild("expire", ExpireSession())
```

Storing Objects in the Session

This example shows you how you can persist objects across requests in the session object.

As was discussed *previously* , instances of Session last as long as the notional session itself does. Each time Request.getSession is called, if the session for the request is still active, then the same `Session` instance is returned as was returned previously. Because of this, `Session` instances can be used to keep other objects around for as long as the session exists.

It's easier to demonstrate how this works than explain it, so here's an example:

```
>>> from zope.interface import Interface, Attribute, implementer
>>> from twisted.python.components import registerAdapter
>>> from twisted.web.server import Session
>>> class ICounter(Interface):
...     value = Attribute("An int value which counts up once per page view.")
...
>>> @implementer(ICounter)
... class Counter(object):
...     def __init__(self, session):
...         self.value = 0
...
>>> registerAdapter(Counter, Session, ICounter)
>>> ses = Session(None, None)
>>> data = ICounter(ses)
>>> print(data)
<__main__.Counter object at 0x8d535ec>
>>> print(data is ICounter(ses))
True
>>>
```

What? , I hear you say.

What's shown in this example is the interface and adaption-based API which `Session` exposes for persisting state. There are several critical pieces interacting here:

- `ICounter` is an interface which serves several purposes. Like all interfaces, it documents the API of some class of objects (in this case, just the `value` attribute). It also serves as a key into what is basically a dictionary within the session object: the interface is used to store or retrieve a value on the session (the `Counter` instance, in this case).
- `Counter` is the class which actually holds the session data in this example. It implements `ICounter` (again, mostly for documentation purposes). It also has a `value` attribute, as the interface declared.
- The registerAdapter call sets up the relationship between its three arguments so that adaption will do what we want in this case.
- Adaption is performed by the expression `ICounter(ses)` . This is read as : adapt `ses` to `ICounter` . Because of the `registerAdapter` call, it is roughly equivalent to `Counter(ses)` . However (because of certain things `Session` does), it also saves the `Counter` instance created so that it will be returned the next time this adaption is done. This is why the last statement produces `True` .

If you're still not clear on some of the details there, don't worry about it and just remember this: `ICounter(ses)` gives you an object you can persist state on. It can be as much or as little state as you want, and you can use as few or as many different `Interface` classes as you want on a single `Session` instance.

With those conceptual dependencies out of the way, it's a very short step to actually getting persistent state into a Twisted Web application. Here's an example which implements a simple counter, re-using the definitions from the example above:

```
from twisted.web.resource import Resource

class CounterResource(Resource):
    def render_GET(self, request):
        session = request.getSession()
        counter = ICounter(session)
        counter.value += 1
        return "Visit #%d for you!" % (counter.value,)
```

Pretty simple from this side, eh? All this does is use `Request.getSession` and the adaption from above, plus

some integer math to give you a session-based visit counter.

Here's the complete source for an *rpy script* based on this example:

```
cache()

from zope.interface import Interface, Attribute, implementer
from twisted.python.components import registerAdapter
from twisted.web.server import Session
from twisted.web.resource import Resource

class ICounter(Interface):
    value = Attribute("An int value which counts up once per page view.")

@implementer(ICounter)
class Counter(object):
    def __init__(self, session):
        self.value = 0

registerAdapter(Counter, Session, ICounter)

class CounterResource(Resource):
    def render_GET(self, request):
        session = request.getSession()
        counter = ICounter(session)
        counter.value += 1
        return "Visit #%d for you!" % (counter.value,)

resource = CounterResource()
```

One more thing to note is the `cache()` call at the top of this example. As with the *previous example* where this came up, this rpy script is stateful. This time, it's the `ICounter` definition and the `registerAdapter` call that need to be executed only once. If we didn't use `cache` , every request would define a new, different interface named `ICounter` . Each of these would be a different key in the session, so the counter would never get past one.

Session Endings

The previous two examples introduced Twisted Web's session APIs. This included accessing the session object, storing state on it, and retrieving it later, as well as the idea that the Session object has a lifetime which is tied to the notional session it represents. This example demonstrates how to exert some control over that lifetime and react when it expires.

The lifetime of a session is controlled by the `sessionTimeout` attribute of the `Session` class. This attribute gives the number of seconds a session may go without being accessed before it expires. The default is 15 minutes. In this example we'll change that to a different value.

One way to override the value is with a subclass:

```
from twisted.web.server import Session

class ShortSession(Session):
    sessionTimeout = 60
```

To have Twisted Web actually make use of this session class, rather than the default, it is also necessary to override the `sessionFactory` attribute of Site . We could do this with another subclass, but we could also do it to just one instance of `Site` :

```
from twisted.web.server import Site
```

```
factory = Site(rootResource)
factory.sessionFactory = ShortSession
```

Sessions given out for requests served by this `Site` will use `ShortSession` and only last one minute without activity.

You can have arbitrary functions run when sessions expire, too. This can be useful for cleaning up external resources associated with the session, tracking usage statistics, and more. This functionality is provided via Session.notifyOnExpire . It accepts a single argument: a function to call when the session expires. Here's a trivial example which prints a message whenever a session expires:

```
from twisted.web.resource import Resource

class ExpirationLogger(Resource):
    sessions = set()

    def render_GET(self, request):
        session = request.getSession()
        if session.uid not in self.sessions:
            self.sessions.add(session.uid)
            session.notifyOnExpire(lambda: self._expired(session.uid))
        return ""

    def _expired(self, uid):
        print("Session", uid, "has expired.")
        self.sessions.remove(uid)
```

Keep in mind that using a method as the callback will keep the instance (in this case, the `ExpirationLogger` resource) in memory until the session expires.

With those pieces in hand, here's an example that prints a message whenever a session expires, and uses sessions which last for 5 seconds:

```
from twisted.web.server import Site, Session
from twisted.web.resource import Resource
from twisted.internet import reactor, endpoints

class ShortSession(Session):
    sessionTimeout = 5

class ExpirationLogger(Resource):
    sessions = set()

    def render_GET(self, request):
        session = request.getSession()
        if session.uid not in self.sessions:
            self.sessions.add(session.uid)
            session.notifyOnExpire(lambda: self._expired(session.uid))
        return ""

    def _expired(self, uid):
        print("Session", uid, "has expired.")
        self.sessions.remove(uid)

rootResource = Resource()
rootResource.putChild("logme", ExpirationLogger())
factory = Site(rootResource)
factory.sessionFactory = ShortSession
```

```
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8080)
endpoint.listen(factory)
reactor.run()
```

Since `Site` customization is required, this example can't be rpy-based, so it brings back the manual `endpoints.TCP4ServerEndpoint` and `reactor.run` calls. Run it and visit `/logme` to see it in action. Keep visiting it to keep your session active. Stop visiting it for five seconds to see your session expiration message.

This set of examples contains short, complete applications of twisted.web. For subjects not covered here, see the *Twisted Web tutorial* and the API documentation.

1. *Serving static content from a directory*
2. *Generating a page dynamically*
3. *Static URL dispatch*
4. *Dynamic URL dispatch*
5. *Error handling*
6. *Custom response codes*
7. *Handling POSTs*
8. *Other request bodies*
9. *rpy scripts (or, how to save yourself some typing)*
10. *Asynchronous responses*
11. *Asynchronous responses (via Deferred)*
12. *Interrupted responses*
13. *Logging errors*
14. *Access logging*
15. *WSGIs*
16. *HTTP authentication*
17. *Session basics*
18. *Storing objects in the session*
19. *Session endings*

Light Weight Templating With Resource Templates

Overview

While high-level templating systems can be used with Twisted (for example, DivmodNevow , sometimes one needs a less file-heavy system which lets one directly write HTML. While ResourceScript is available, it has a high coding overhead, and requires some boring string arithmetic. ResourceTemplate fills the space between Nevow and ResourceScript using Quixote's PTL (Python Templating Language).

ResourceTemplates need Quixote installed. In Debian , that means installing the `python-quixote` package (`apt-get install python-quixote`). Other operating systems require other ways to install Quixote, or it can be done manually.

Configuring Twisted Web

The easiest way to get Twisted Web to support ResourceTemplates is to bind them to some extension using the web tap's `--processor` flag. Here is an example:

```
% twistd web --path=/var/www \
        --processor=.rtl=twisted.web.script.ResourceTemplate
```

The above command line binds the `rtl` extension to use the ResourceTemplate processor. Other ways are possible, but would require more Python coding and are outside the scope of this HOWTO.

Using ResourceTemplate

ResourceTemplates are coded in an extension of Python called the"Python Templating Language" . Complete documentation of the PTL is available at the quixote web site . The web server will expect the PTL source file to define a variable named `resource` . This should be a twisted.web.resource.Resource , whose `.render` method be called. Usually, you would want to define `render` using the keyword `template` rather than `def` .

Here is a simple example for a resource template.

`webquote.rtl`

```
from twisted.web.resource import Resource

def getQuote():
    return "An apple a day keeps the doctor away."


class QuoteResource(Resource):

    template render(self, request):
        """\
        <html>
        <head><title>Quotes Galore</title></head>

        <body><h1>Quotes</h1>"""
        getQuote()
        "</body></html>"


resource = QuoteResource()
```

Using the Twisted Web Client

Overview

This document describes how to use the HTTP client included in Twisted Web. After reading it, you should be able to make HTTP and HTTPS requests using Twisted Web. You will be able to specify the request method, headers, and body and you will be able to retrieve the response code, headers, and body.

A number of higher-level features are also explained, including proxying, automatic content encoding negotiation, and cookie handling.

Prerequisites

This document assumes that you are familiar with *Deferreds and Failures* , and *producers and consumers* . It also assumes you are familiar with the basic concepts of HTTP, such as requests and responses, methods, headers, and message bodies. The HTTPS section of this document also assumes you are somewhat familiar with SSL and have read about *using SSL in Twisted* .

The Agent

Issuing Requests

The twisted.web.client.Agent class is the entry point into the client API. Requests are issued using the request method, which takes as parameters a request method, a request URI, the request headers, and an object which can produce the request body (if there is to be one). The agent is responsible for connection setup. Because of this, it requires a reactor as an argument to its initializer. An example of creating an agent and issuing a request using it might look like this:

`request.py`

```
from __future__ import print_function

from twisted.internet import reactor
from twisted.web.client import Agent
from twisted.web.http_headers import Headers

agent = Agent(reactor)

d = agent.request(
    b'GET',
    b'http://httpbin.com/anything',
    Headers({'User-Agent': ['Twisted Web Client Example']}),
    None)

def cbResponse(ignored):
    print('Response received')
d.addCallback(cbResponse)

def cbShutdown(ignored):
    reactor.stop()
d.addBoth(cbShutdown)

reactor.run()
```

As may be obvious, this issues a new *GET* request for / to the web server on `example.com` . `Agent` is responsible for resolving the hostname into an IP address and connecting to it on port 80 (for *HTTP* URIs), port 443 (for *HTTPS* URIs), or on the port number specified in the URI itself. It is also responsible for cleaning up the connection afterwards. This code sends a request which includes one custom header, *User-Agent* . The last argument passed to `Agent.request` is `None` , though, so the request has no body.

Sending a request which does include a body requires passing an object providing twisted.web.iweb.IBodyProducer to `Agent.request` . This interface extends the more general IPushProducer by adding a new `length` attribute and adding several constraints to the way the producer and consumer interact.

- The length attribute must be a non-negative integer or the constant `twisted.web.iweb.UNKNOWN_LENGTH` . If the length is known, it will be used to specify the value for the *Content-Length* header in the request. If the length is unknown the attribute should be set to `UNKNOWN_LENGTH` . Since more servers support *Content-Length* , if a length can be provided it should be.

- An additional method is required on `IBodyProducer` implementations: `startProducing` . This method is used to associate a consumer with the producer. It should return a `Deferred` which fires when all data has been produced.
- `IBodyProducer` implementations should never call the consumer's `unregisterProducer` method. Instead, when it has produced all of the data it is going to produce, it should only fire the `Deferred` returned by `startProducing` .

For additional details about the requirements of IBodyProducer implementations, see the API documentation.

Here's a simple `IBodyProducer` implementation which writes an in-memory string to the consumer:

`bytesprod.py`

```
from zope.interface import implementer

from twisted.internet.defer import succeed
from twisted.web.iweb import IBodyProducer

@implementer(IBodyProducer)
class BytesProducer(object):
    def __init__(self, body):
        self.body = body
        self.length = len(body)

    def startProducing(self, consumer):
        consumer.write(self.body)
        return succeed(None)

    def pauseProducing(self):
        pass

    def stopProducing(self):
        pass
```

This producer can be used to issue a request with a body:

`sendbody.py`

```
from __future__ import print_function

from twisted.internet import reactor
from twisted.web.client import Agent
from twisted.web.http_headers import Headers

from bytesprod import BytesProducer

agent = Agent(reactor)
body = BytesProducer(b"hello, world")
d = agent.request(
    b'POST',
    b'http://httpbin.org/post',
    Headers({'User-Agent': ['Twisted Web Client Example'],
             'Content-Type': ['text/x-greeting']}),
    body)

def cbResponse(ignored):
    print('Response received')
d.addCallback(cbResponse)
```

```
def cbShutdown(ignored):
    reactor.stop()
d.addBoth(cbShutdown)

reactor.run()
```

If you want to upload a file or you just have some data in a string, you don't have to copy `StringProducer` though. Instead, you can use FileBodyProducer . This `IBodyProducer` implementation works with any file-like object (so use it with a `StringIO` if your upload data is already in memory as a string); the idea is the same as `StringProducer` from the previous example, but with a little extra code to only send data as fast as the server will take it.

`filesendbody.py`

```
from __future__ import print_function

from io import BytesIO

from twisted.internet import reactor
from twisted.web.client import Agent
from twisted.web.http_headers import Headers

from twisted.web.client import FileBodyProducer

agent = Agent(reactor)
body = FileBodyProducer(BytesIO(b"hello, world"))
d = agent.request(
    b'GET',
    b'http://example.com/',
    Headers({'User-Agent': ['Twisted Web Client Example'],
             'Content-Type': ['text/x-greeting']}),
    body)

def cbResponse(ignored):
    print('Response received')
d.addCallback(cbResponse)

def cbShutdown(ignored):
    reactor.stop()
d.addBoth(cbShutdown)

reactor.run()
```

`FileBodyProducer` closes the file when it no longer needs it.

If the connection or the request take too much time, you can cancel the `Deferred` returned by the `Agent.request` method. This will abort the connection, and the `Deferred` will errback with CancelledError .

Receiving Responses

So far, the examples have demonstrated how to issue a request. However, they have ignored the response, except for showing that it is a `Deferred` which seems to fire when the response has been received. Next we'll cover what that response is and how to interpret it.

`Agent.request` , as with most `Deferred` -returning APIs, can return a `Deferred` which fires with a `Failure` . If the request fails somehow, this will be reflected with a failure. This may be due to a problem looking up the host IP address, or it may be because the HTTP server is not accepting connections, or it may be because of a problem

parsing the response, or any other problem which arises which prevents the response from being received. It does *not* include responses with an error status.

If the request succeeds, though, the `Deferred` will fire with a Response . This happens as soon as all the response headers have been received. It happens before any of the response body, if there is one, is processed. The `Response` object has several attributes giving the response information: its code, version, phrase, and headers, as well as the length of the body to expect. In addition to these, the `Response` also contains a reference to the request that it is a response to; one particularly useful attribute on the request is absoluteURI : The absolute URI to which the request was made. The `Response` object has a method which makes the response body available: deliverBody . Using the attributes of the response object and this method, here's an example which displays part of the response to a request:

`response.py`

```
from __future__ import print_function

from pprint import pformat

from twisted.internet import reactor
from twisted.internet.defer import Deferred
from twisted.internet.protocol import Protocol
from twisted.web.client import Agent
from twisted.web.http_headers import Headers

class BeginningPrinter(Protocol):
    def __init__(self, finished):
        self.finished = finished
        self.remaining = 1024 * 10

    def dataReceived(self, bytes):
        if self.remaining:
            display = bytes[:self.remaining]
            print('Some data received:')
            print(display)
            self.remaining -= len(display)

    def connectionLost(self, reason):
        print('Finished receiving body:', reason.getErrorMessage())
        self.finished.callback(None)

agent = Agent(reactor)
d = agent.request(
    b'GET',
    b'http://httpbin.com/anything/',
    Headers({'User-Agent': ['Twisted Web Client Example']}),
    None)

def cbRequest(response):
    print('Response version:', response.version)
    print('Response code:', response.code)
    print('Response phrase:', response.phrase)
    print('Response headers:')
    print(pformat(list(response.headers.getAllRawHeaders())))
    finished = Deferred()
    response.deliverBody(BeginningPrinter(finished))
    return finished
d.addCallback(cbRequest)

def cbShutdown(ignored):
    reactor.stop()
```

```
d.addBoth(cbShutdown)

reactor.run()
```

The `BeginningPrinter` protocol in this example is passed to `Response.deliverBody` and the response body is then delivered to its `dataReceived` method as it arrives. When the body has been completely delivered, the protocol's `connectionLost` method is called. It is important to inspect the `Failure` passed to `connectionLost`. If the response body has been completely received, the failure will wrap a twisted.web.client.ResponseDone exception. This indicates that it is *known* that all data has been received. It is also possible for the failure to wrap a twisted.web.http.PotentialDataLoss exception: this indicates that the server framed the response such that there is no way to know when the entire response body has been received. Only HTTP/1.0 servers should behave this way. Finally, it is possible for the exception to be of another type, indicating guaranteed data loss for some reason (a lost connection, a memory error, etc).

Just as protocols associated with a TCP connection are given a transport, so will be a protocol passed to `deliverBody`. Since it makes no sense to write more data to the connection at this stage of the request, though, the transport *only* provides IPushProducer. This allows the protocol to control the flow of the response data: a call to the transport's `pauseProducing` method will pause delivery; a later call to `resumeProducing` will resume it. If it is decided that the rest of the response body is not desired, `stopProducing` can be used to stop delivery permanently; after this, the protocol's `connectionLost` method will be called.

An important thing to keep in mind is that the body will only be read from the connection after `Response.deliverBody` is called. This also means that the connection will remain open until this is done (and the body read). So, in general, any response with a body *must* have that body read using `deliverBody`. If the application is not interested in the body, it should issue a *HEAD* request or use a protocol which immediately calls `stopProducing` on its transport.

If the body of the response isn't going to be consumed incrementally, then readBody can be used to get the body as a byte-string. This function returns a `Deferred` that fires with the body after the request has been completed; cancelling this `Deferred` will close the connection to the HTTP server immediately.

`responseBody.py`

```
from __future__ import print_function

from sys import argv
from pprint import pformat

from twisted.internet.task import react
from twisted.web.client import Agent, readBody
from twisted.web.http_headers import Headers


def cbRequest(response):
    print('Response version:', response.version)
    print('Response code:', response.code)
    print('Response phrase:', response.phrase)
    print('Response headers:')
    print(pformat(list(response.headers.getAllRawHeaders())))
    d = readBody(response)
    d.addCallback(cbBody)
    return d

def cbBody(body):
    print('Response body:')
    print(body)

def main(reactor, url=b"http://httpbin.org/get"):
```

```
    agent = Agent(reactor)
    d = agent.request(
        b'GET', url,
        Headers({'User-Agent': ['Twisted Web Client Example']}),
        None)
    d.addCallback(cbRequest)
    return d

react(main, argv[1:])
```

Customizing your HTTPS Configuration

Everything you've read so far applies whether the scheme of the request URI is *HTTP* or *HTTPS* .

Since version 15.0.0, Twisted's `Agent` HTTP client will validate `https` URLs against your platform's trust store by default.

Note: If you're using an earlier version, please upgrade, as this is a *critical* security feature!

Note: Only `Agent`, and things that use it, validates HTTPS. Do not use `getPage` to retrieve HTTPS URLs; although we have not yet removed all the examples that use it, we discourage its use.

You should `pip install twisted[tls]` in order to get all the dependencies necessary to do TLS properly. The *installation instructions* gives more detail on optional dependencies and how to install them which may be of interest.

For some uses, you may need to customize Agent's use of HTTPS; for example, to provide a client certificate, or to use a custom certificate authority for an internal network.

Here, we're just going to show you how to inject the relevant TLS configuration into an Agent, so we'll use the simplest possible example, rather than a more useful but complex one.

`Agent`'s constructor takes an optional second argument, which allows you to customize its behavior with respect to HTTPS. The object passed here must provide the twisted.web.iweb.IPolicyForHTTPS interface.

Using the very helpful `badssl.com` web API, we will construct a request that fails to validate, because the certificate has the wrong hostname in it. The following Python program should produce a verification error when run as `python example.py https://wrong.host.badssl.com/`.

```
import sys

from twisted.internet.task import react

from twisted.web.client import Agent, ResponseFailed

@react
def main(reactor):
    agent = Agent(reactor)
    requested = agent.request(b"GET", sys.argv[1].encode("ascii"))
    def gotResponse(response):
        print(response.code)
    def noResponse(failure):
        failure.trap(ResponseFailed)
        print(failure.value.reasons[0].getTraceback())
    return requested.addCallbacks(gotResponse, noResponse)
```

The verification error you see when running the above program is due to the mismatch between "wrong.host.badssl.com", the expected hostname derived from the URL, and "badssl.com", the hostname contained in the certificate presented by the server. In our pretend scenario, we want to construct an Agent that validates HTTPS certificates normally, _except_ for this one host. For "wrong.host.badssl.com", we wish to supply the correct hostname to check the certificate against manually, to work around this problem. In order to do that, we will supply our own policy that creates a different TLS configuration depending on the hostname, like so:

```
import sys

from zope.interface import implementer

from twisted.internet.task import react
from twisted.internet.ssl import optionsForClientTLS

from twisted.web.iweb import IPolicyForHTTPS
from twisted.web.client import Agent, ResponseFailed, BrowserLikePolicyForHTTPS

@implementer(IPolicyForHTTPS)
class OneHostnameWorkaroundPolicy(object):
    def __init__(self):
        self._normalPolicy = BrowserLikePolicyForHTTPS()
    def creatorForNetloc(self, hostname, port):
        if hostname == b"wrong.host.badssl.com":
            hostname = b"badssl.com"
        return self._normalPolicy.creatorForNetloc(hostname, port)

@react
def main(reactor):
    agent = Agent(reactor, OneHostnameWorkaroundPolicy())
    requested = agent.request(b"GET", sys.argv[1].encode("ascii"))
    def gotResponse(response):
        print(response.code)
    def noResponse(failure):
        failure.trap(ResponseFailed)
        print(failure.value.reasons[0].getTraceback())
    return requested.addCallbacks(gotResponse, noResponse)
```

Now, invoking `python example.py https://wrong.host.badssl.com/` will happily give us a `200` status code; however, running it with `https://expired.badssl.com/` or `https://self-signed.badssl.com/` or any of the other error hostnames should still give an error.

Note: The technique presented above does not *ignore the mismatching hostname*, but rather, it *provides the erroneous hostname specifically* so that the misconfiguration is expected. Twisted does not document any facilities for disabling verification, since that makes TLS useless; instead, we strongly encourage you to figure out what properties you need from the connection and verify those.

Using this TLS policy mechanism, you can customize Agent to use any feature of TLS that Twisted has support for, including the examples given above; client certificates, alternate trust roots, and so on. For a more detailed explanation of what options exist for client TLS configuration in Twisted, check out the documentation for the optionsForClientTLS API and the *Using SSL in Twisted* chapter of this documentation.

HTTP Persistent Connection

HTTP persistent connections use the same TCP connection to send and receive multiple HTTP requests/responses. This reduces latency and TCP connection establishment overhead.

The constructor of twisted.web.client.Agent takes an optional parameter pool, which should be an instance of HTTPConnectionPool , which will be used to manage the connections. If the pool is created with the parameter `persistent` set to `True` (the default), it will not close connections when the request is done, and instead hold them in its cache to be re-used.

Here's an example which sends requests over a persistent connection:

```
from twisted.internet import reactor
from twisted.internet.defer import Deferred, DeferredList
from twisted.internet.protocol import Protocol
from twisted.web.client import Agent, HTTPConnectionPool

class IgnoreBody(Protocol):
    def __init__(self, deferred):
        self.deferred = deferred

    def dataReceived(self, bytes):
        pass

    def connectionLost(self, reason):
        self.deferred.callback(None)


def cbRequest(response):
    print('Response code:', response.code)
    finished = Deferred()
    response.deliverBody(IgnoreBody(finished))
    return finished

pool = HTTPConnectionPool(reactor)
agent = Agent(reactor, pool=pool)

def requestGet(url):
    d = agent.request('GET', url)
    d.addCallback(cbRequest)
    return d

# Two requests to the same host:
d = requestGet('http://localhost:8080/foo').addCallback(
    lambda ign: requestGet("http://localhost:8080/bar"))
def cbShutdown(ignored):
    reactor.stop()
d.addCallback(cbShutdown)

reactor.run()
```

Here, the two requests are to the same host, one after the each other. In most cases, the same connection will be used for the second request, instead of two different connections when using a non-persistent pool.

Multiple Connections to the Same Server

twisted.web.client.HTTPConnectionPool instances have an attribute called `maxPersistentPerHost` which limits the number of cached persistent connections to the same server. The default value is 2. This is effective only when the persistent option is True. You can change the value like bellow:

```
from twisted.web.client import HTTPConnectionPool
```

```
pool = HTTPConnectionPool(reactor, persistent=True)
pool.maxPersistentPerHost = 1
```

With the default value of 2, the pool keeps around two connections to the same host at most. Eventually the cached persistent connections will be closed, by default after 240 seconds; you can change this timeout value with the `cachedConnectionTimeout` attribute of the pool. To force all connections to close use the closeCachedConnections method.

Automatic Retries

If a request fails without getting a response, and the request is something that hopefully can be retried without having any side-effects (e.g. a request with method GET), it will be retried automatically when sending a request over a previously-cached persistent connection. You can disable this behavior by setting retryAutomatically to `False` . Note that each request will only be retried once.

Following redirects

By itself, `Agent` doesn't follow HTTP redirects (responses with 301, 302, 303, 307 status codes and a `location` header field). You need to use the twisted.web.client.RedirectAgent class to do so. It implements a rather strict behavior of the RFC, meaning it will redirect 301 and 302 as 307, only on `GET` and `HEAD` requests.

The following example shows how to have a redirect-enabled agent.

```
from twisted.python.log import err
from twisted.web.client import Agent, RedirectAgent
from twisted.internet import reactor

def display(response):
    print("Received response")
    print(response)

def main():
    agent = RedirectAgent(Agent(reactor))
    d = agent.request("GET", "http://example.com/")
    d.addCallbacks(display, err)
    d.addCallback(lambda ignored: reactor.stop())
    reactor.run()

if __name__ == "__main__":
    main()
```

In contrast, twisted.web.client.BrowserLikeRedirectAgent implements more lenient behaviour that closely emulates what web browsers do; in other words 301 and 302 `POST` redirects are treated like 303, meaning the method is changed to `GET` before making the redirect request.

As mentioned previously, Response contains a reference to both the request that it is a response to, and the previously received response , accessible by previousResponse . In most cases there will not be a previous response, but in the case of `RedirectAgent` the response history can be obtained by following the previous responses from response to response.

Using a HTTP proxy

To be able to use HTTP proxies with an agent, you can use the twisted.web.client.ProxyAgent class. It supports the same interface as `Agent`, but takes the endpoint of the proxy as initializer argument. This is specifically intended for talking to servers that implement the proxying variation of the HTTP protocol; for other types of proxies you will want Agent.usingEndpointFactory (see documentation below).

Here's an example demonstrating the use of an HTTP proxy running on localhost:8000.

```
from twisted.python.log import err
from twisted.web.client import ProxyAgent
from twisted.internet import reactor
from twisted.internet.endpoints import TCP4ClientEndpoint

def display(response):
    print("Received response")
    print(response)

def main():
    endpoint = TCP4ClientEndpoint(reactor, "localhost", 8000)
    agent = ProxyAgent(endpoint)
    d = agent.request("GET", "https://example.com/")
    d.addCallbacks(display, err)
    d.addCallback(lambda ignored: reactor.stop())
    reactor.run()

if __name__ == "__main__":
    main()
```

Please refer to the *endpoints documentation* for more information about how they work and the twisted.internet.endpoints API documentation to learn what other kinds of endpoints exist.

Handling HTTP cookies

An existing agent instance can be wrapped with twisted.web.client.CookieAgent to automatically store, send and track HTTP cookies. A `CookieJar` instance, from the Python standard library module cookielib , is used to store the cookie information. An example of using `CookieAgent` to perform a request and display the collected cookies might look like this:

`cookies.py`

```
from __future__ import print_function

from twisted.internet import reactor
from twisted.python import log, compat
from twisted.web.client import Agent, CookieAgent

def displayCookies(response, cookieJar):
    print('Received response')
    print(response)
    print('Cookies:', len(cookieJar))
    for cookie in cookieJar:
        print(cookie)

def main():
    cookieJar = compat.cookielib.CookieJar()
    agent = CookieAgent(Agent(reactor), cookieJar)
```

```
    d = agent.request(b'GET', b'http://httpbin.org/cookies/set?some=data')
    d.addCallback(displayCookies, cookieJar)
    d.addErrback(log.err)
    d.addCallback(lambda ignored: reactor.stop())
    reactor.run()

if __name__ == "__main__":
    main()
```

Automatic Content Encoding Negotiation

twisted.web.client.ContentDecoderAgent adds support for sending *Accept-Encoding* request headers and interpreting *Content-Encoding* response headers. These headers allow the server to encode the response body somehow, typically with some compression scheme to save on transfer costs. `ContentDecoderAgent` provides this functionality as a wrapper around an existing agent instance. Together with one or more decoder objects (such as twisted.web.client.GzipDecoder), this wrapper automatically negotiates an encoding to use and decodes the response body accordingly. To application code using such an agent, there is no visible difference in the data delivered.

`gzipdecoder.py`

```
from __future__ import print_function

from twisted.python import log
from twisted.internet import reactor
from twisted.internet.defer import Deferred
from twisted.internet.protocol import Protocol
from twisted.web.client import Agent, ContentDecoderAgent, GzipDecoder

class BeginningPrinter(Protocol):
    def __init__(self, finished):
        self.finished = finished
        self.remaining = 1024 * 10


    def dataReceived(self, bytes):
        if self.remaining:
            display = bytes[:self.remaining]
            print('Some data received:')
            print(display)
            self.remaining -= len(display)


    def connectionLost(self, reason):
        print('Finished receiving body:', reason.type, reason.value)
        self.finished.callback(None)



def printBody(response):
    finished = Deferred()
    response.deliverBody(BeginningPrinter(finished))
    return finished


def main():
```

```
    agent = ContentDecoderAgent(Agent(reactor), [(b'gzip', GzipDecoder)])

    d = agent.request(b'GET', b'http://httpbin.org/gzip')
    d.addCallback(printBody)
    d.addErrback(log.err)
    d.addCallback(lambda ignored: reactor.stop())
    reactor.run()

if __name__ == "__main__":
    main()
```

Implementing support for new content encodings is as simple as writing a new class like `GzipDecoder` that can decode a response using the new encoding. As there are not many content encodings in widespread use, gzip is the only encoding supported by Twisted itself.

Connecting To Non-standard Destinations

Typically you want your HTTP client to open a TCP connection directly to the web server. Sometimes however it's useful to be able to connect some other way, e.g. making an HTTP request over a SOCKS proxy connection or connecting to a server listening on a UNIX socket. For this reason, there is an alternate constructor called Agent.usingEndpointFactory that takes an `endpointFactory` argument. This argument must provide the twisted.web.iweb.IAgentEndpointFactory interface. Note that when talking to a HTTP proxy, i.e. a server that implements the proxying-specific variant of HTTP you should use ProxyAgent - see documentation above.

`endpointconstructor.py`

```
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

"""
Send a HTTP request to Docker over a Unix socket.

Will probably need to be run as root.

Usage:
    $ sudo python endpointconstructor.py [<docker API path>]
"""

from __future__ import print_function

from sys import argv

from zope.interface import implementer

from twisted.internet.endpoints import UNIXClientEndpoint
from twisted.internet.task import react
from twisted.web.iweb import IAgentEndpointFactory
from twisted.web.client import Agent, readBody


@implementer(IAgentEndpointFactory)
class DockerEndpointFactory(object):
    """
    Connect to Docker's Unix socket.
    """
    def __init__(self, reactor):
```

```
        self.reactor = reactor


    def endpointForURI(self, uri):
        return UNIXClientEndpoint(self.reactor, b"/var/run/docker.sock")



def main(reactor, path=b"/containers/json?all=1"):
    agent = Agent.usingEndpointFactory(reactor, DockerEndpointFactory(reactor))
    d = agent.request(b'GET', b"unix://localhost" + path)
    d.addCallback(readBody)
    d.addCallback(print)
    return d

react(main, argv[1:])
```

Conclusion

You should now understand the basics of the Twisted Web HTTP client. In particular, you should understand:

- How to issue requests with arbitrary methods, headers, and bodies.
- How to access the response version, code, phrase, headers, and body.
- How to store, send, and track cookies.
- How to control the streaming of the response body.
- How to enable the HTTP persistent connection, and control the number of connections.

Glossary

This glossary is very incomplete. Contributions are welcome. resource

An object accessible via HTTP at one or more URIs. In Twisted Web, a resource is represented by an object which provides twisted.web.resource.IResource and most often is a subclass of twisted.web.resource.Resource . For example, here is a resource which represents a simple HTML greeting.

```
from twisted.web.resource import Resource

class Greeting(Resource):
    def render_GET(self, request):
        return "
```

- Introduction
  - *Overview of Twisted Web*
- Web Applications
  - *Using twisted.web*
  - *Web application development*
  - *HTML Templating with twisted.web.template*
  - *XML-RPC and SOAP*

Examples

twisted.web.client

- `httpclient.py` - use `twisted.web.client.Agent` to download a web page.
- (deprecated) `getpage.py` - use `twisted.web.client.getPage` to download a web page.
- (deprecated) `dlpage.py` - add callbacks to `twisted.web.client.downloadPage` to display errors that occur when downloading a web page

XML-RPC

- `xmlrpc.py` XML-RPC server with several methods, including echoing, faulting, returning deferreds and failed deferreds
- `xmlrpcclient.py` - use `twisted.web.xmlrpc.Proxy` to call remote XML-RPC methods
- `advogato.py` - use `twisted.web.xmlrpc` to post a diary entry to advogato.org; requires an advogato account

Virtual hosts and proxies

- `proxy.py` - use `twisted.web.proxy.Proxy` to make the simplest proxy
- `logging-proxy.py` - example of subclassing the core classes of `twisted.web.proxy` to log requests through a proxy
- `reverse-proxy.py` - use `twisted.web.proxy.ReverseProxyResource` to make any HTTP request to the proxy port get applied to a specified website
- `rootscript.py` - example use of `twisted.web.vhost.NameVirtualHost`
- `web.py` - an example of both using the `processors` attribute to set how certain file types are treated and using `twisted.web.vhost.VHostMonsterResource` to reverse proxy

.rpys and ResourceTemplate

- `hello.rpy.py` - use `twisted.web.static` to create a static resource to serve
- `fortune.rpy.py` - create a resource that returns the output of a process run on the server
- `report.rpy.py` - display various properties of a resource, including path, host, and port
- `users.rpy.py` - use `twisted.web.distrib` to publish user directories as for a "community web site"

- `simple.rtl` - example use of `twisted.web.resource.ResourceTemplate`

Miscellaneous

- `webguard.py` - pairing `twisted.web` with `twisted.cred` to guard resources against unauthenticated users
- `silly-web.py` - bare-bones distributed web setup with a master and slave using `twisted.web.distrib` and `twisted.spread.pb`
- `soap.py` - use `twisted.web.soap` to publish SOAP methods
- *Developer guides*: documentation on using Twisted Web to develop your own applications
- *Examples*: short code examples using Twisted Web

CHAPTER 8

Twisted Words

Developer Guides

Overview of Twisted IM

Twisted IM (Instance Messenger) is a multi-protocol chat framework, based on the Twisted framework we've all come to know and love. It's fairly simple and extensible in two directions - it's pretty easy to add new protocols, and it's also quite easy to add new front-ends.

Code flow

AccountManager

The control flow starts at the relevant subclass of baseaccount.AccountManager . The AccountManager is responsible for, well, managing accounts - remembering what accounts are available, their settings, adding and removal of accounts, and making accounts log on at startup.

This would be a good place to start your interface, load a list of accounts from disk and tell them to login. Most of the method names in AccountManager are pretty self-explanatory, and your subclass can override whatever it wants, but you *need* to override `__init__` . Something like this:

```
...
    def __init__(self):
        self.chatui = ... # Your subclass of basechat.ChatUI
        self.accounts = ... # Load account list
        for a in self.accounts:
            a.logOn(self.chatui)
```

ChatUI

Account objects talk to the user via a subclass of basechat.ChatUI . This class keeps track of all the various conversations that are currently active, so that when an account receives and incoming message, it can put that message in its correct context.

How much of this class you need to override depends on what you need to do. You will need to override `getConversation` (a one-on-one conversation, like an IRC DCC chat) and `getGroupConversation` (a multiple user conversation, like an IRC channel). You might want to override `getGroup` and `getPerson` .

The main problem with the default versions of the above routines is that they take a parameter, `Class` , which defaults to an abstract implementation of that class - for example, `getConversation` has a `Class` parameter that defaults to basechat.Conversation which raises a lot of `NotImplementedError` s. In your subclass, override the method with a new method whose Class parameter defaults to your own implementation of `Conversation` , that simply calls the parent class' implementation.

Conversation and GroupConversation

These classes are where your interface meets the chat protocol. Chat protocols get a message, find the appropriate `Conversation` or `GroupConversation` object, and call its methods when various interesting things happen.

Override whatever methods you want to get the information you want to display. You must override the `hide` and `show` methods, however - they are called frequently and the default implementation raises `NotImplementedError` .

Accounts

An account is an instance of a subclass of basesupport.AbstractAccount . For more details and sample code, see the various `*support` files in `twisted.words.im` .

Using the Twisted IRC Client

A complete howto would explain how to actually use the IRC client. However, until that howto is written, here is a howto that explains how to do text formatting for IRC.

Text formatting

The text formatting support in Twisted Words is based on the widely used mIRC format which supports bold, underline, reverse video and colored text; nesting these attributes is also supported.

Creating formatted text

The API used for creating formatted text in the IRC client is almost the same as that used by Twisted insults . Text attributes are built up by accessing and indexing attributes on a special module-level attribute, twisted.words.protocols.irc.attributes , multiple values can be passed when indexing attributes to mix text with nested text attributes. The resulting object can then be serialized to formatted text, with twisted.words.protocols.irc.assembleFormattedText , suitable for use with any of the IRC client messaging functions.

Bold, underline and reverse video attributes

Bold, underline and reverse video attributes are just flags and are the simplest text attributes to apply. They are accessed by the names `bold`, `underline` and `reverseVideo`, respectively, on twisted.words.protocols.irc.attributes. For example, messaging someone the bold and underlined text "Hello world!":

```
from twisted.words.protocols.irc import assembleFormattedText, attributes as A

# Message "someone" the bold and underlined text "Hello world!"
anIRCClient.msg('someone', assembleFormattedText(
    A.bold[
        A.underline['Hello world!']])
```

The "normal" attribute

At first glance a text attribute called "normal" that does not apply any unusual text attributes may not seem that special but it can be quite useful, both as a container:

```
A.normal[
    'This is normal text. ',
    A.bold['This is bold text! '],
    'Back to normal',
    A.underline['This is underlined text!']]
```

And also as a way to temporarily disable text attributes without having to close and respecify all text attributes for a brief piece of text:

```
A.normal[
    A.reverseVideo['This is reverse, ', A.normal['except for this'], ', text']]
```

It is worth noting that assembled text will always begin with the control code to disable other attributes for the sake of correctness.

Color attributes

Since colors for both the foreground and background can be specified with IRC text formatting another level of attribute access is introduced. Firstly the foreground or background, through the `fg` and `bg` attribute names respectively, is accessed and then the color name is accessed. The available color attribute names are:

- white
- black
- blue
- green
- lightRed
- red
- magenta
- orange
- yellow
- lightGreen

- cyan
- lightCyan
- lightBlue
- lightMagenta
- gray
- lightGray

It is possible to nest foreground and background colors to alter both for a single piece of text. For example to display black on green text:

```
A.fg.black[A.bg.green['Like a terminal!']]
```

Parsing formatted text

Most IRC clients format text so it is logical that you may want to parse this formatted text. twisted.words.protocols.irc.parseFormattedText will parse text into structured text attributes. It is worth noting that while feeding the output of `parseFormattedText` back to `assembleFormattedText` will produce the same final result, the actual structure of the parsed text will differ. Color codes are mapped from 0 to 15, codes greater than 15 will begin to wrap around.

Removing formatting

In some cases, such as an automaton handling user input from IRC, it is desirable to have all formatting stripped from text. This can be accomplished with twisted.words.protocols.irc.stripFormatting .

Communicating With IRC Clients

Communicating with clients is the whole point of an IRC server, so you want to make sure you're doing it properly. Today, we'll be looking at receiving messages from a client and sending messages to the client.

Representing Clients in Twisted

Users in Twisted IRC are represented as subclasses of the IRC class. This works as the protocol for your Factory class. It will also give you IRC features (like automatically parsing incoming lines) without you having to implement them yourself. The rest of this guide assumes this setup.

Sending Messages

Messages are sent to users using the user object's sendMessage method.

Sending Basic Messages

The basic syntax for sending messages to users is as follows:

```
user.sendCommand("COMMAND", (param1, param2), server.name)
```

The prefix keyword argument is optional, and it may be omitted to send a message without a prefix (for example, the ERROR command). The command is whatever command you plan to send, e.g. “PRIVMSG”, “MODE”, etc. All arguments following the command are the parameters you want to send for the command. If the last argument needs to be prefixed with a colon (because it has spaces in it, e.g. a PRIVMSG message), you must add the colon to the beginning of the parameter yourself. For example: .. code-block:: python

user.sendCommand(“PRIVMSG”, (user.nickname, ”:{}”.format(message)), sendingUser.hostmask)

Sending Messages with Tags

Twisted also allows sending message tags as specified in IRCv3.

Let’s say, for example, that your server has a feature to play back a little bit of previous channel content when someone joins a channel. You want a way to tell people when this message occurred. The best way to provide this information is through the server-time specification.

Let’s say you’re storing past messages in a channel object in some structure like this:

```
channel.pastMessages = [
    ("I sent some text!", "author!ident@host", datetime object representing the when␣
↪the message was sent),
    ("I did, too!", "someone-else!ident@host", another datetime object)
]
```

Your actual implementation may vary. I went with something simple here. The times of the messages would be generated using something like `datetime.utcnow()` when the message was received.

Tags are passed as a list of tuples. If you’re sending a number of tags, you may have an existing tag dictionary. You can simply add to it (assuming `message` is the loop variable for channel.pastMessages above):

```
sendingTags["server-time"] = "{}Z".format(message[2].isoformat()[:-3])
```

This will generate the required time format and add it to the tag dictionary. The last three characters that we remove are the microseconds; removing the last three digits changes the precision to milliseconds.

Once your tags are collected, you can send the message. The tag dictionary is passed using the `tags` argument (in the same loop as above):

```
user.sendCommand("PRIVMSG", (user.nickname, message[0]), message[1], sendingTags)
```

Receiving Messages

Twisted Words will handle receiving messages and parsing lines into tokens. The parsed messages are passed into your command through the user’s handleCommand method.

Handling Commands

The default IRC handleCommand method calls the `irc_COMMAND` method when it receives the command `COMMAND`, and it calls irc_unknown if the method for the command received isn’t defined.

```
from twisted.words.protocols import irc

class IRCUser(irc.IRC):
    # possibly other definitions here
    def irc_unknown(self, prefix, command, params):
```

```
        self.sendCommand(irc.ERR_UNKNOWNCOMMAND, (command, ":Unknown command"),␣
↪server.name)

    def irc_PRIVMSG(self, prefix, params):
        # do some stuff to handle PRIVMSG for your server's setup

    # lots of other command definitions
```

If you have a server setup that doesn't allow you to do this (e.g. a modular server program), you may, of course, override the handleCommand function to route commands to your own handlers.

Receiving Messages with Tags

This has not yet been implemented.

- *Twisted IM*
- IRC
  - *Using the Twisted Words IRC client*
  - *IRC Servers: Communicating With Clients*

Examples

- `ircLogBot.py` - connects to an IRC server and logs all messages
- `minchat.py` - log bot using twisted.im
- `pb_client.py`
- `xmpp_client.py`
- `cursesclient.py` - trivial curses-based IRC client
- *Developer guides*: documentation on using Twisted Words to develop your own applications
- *Examples*: short code examples using Twisted Words

CHAPTER 9

Historical Documents

Here are documents which contain no pertinent information or documentation. People from the Twisted team have published them, and they serve as interesting land marks and thoughts. Please don't look here for documentation – however, if you are interested in the history of Twisted, or want to quote from these documents, feel free. Remember, however – the documents here may contain wrong information – they are not updated as Twisted is, to keep their historical value intact.

2003

Python Community Conference

These papers were part of the Python Community Conference (PyCon) in March of 2003.

Generalization of Deferred Execution in Python

> A deceptively simple architectural challenge faced by many multi-tasking applications is gracefully doing nothing. Systems that must wait for the results of a long-running process, network message, or database query while continuing to perform other tasks must establish conventions for the semantics of waiting. The simplest of these is blocking in a thread, but it has significant scalability problems. In asynchronous frameworks, the most common approach is for long-running methods to accept a callback that will be executed when the command completes. These callbacks will have different signatures depending on the nature of the data being requested, and often, a great deal of code is necessary to glue one portion of an asynchronous networking system to another. Matters become even more complicated when a developer wants to wait for two different events to complete, requiring the developer to "juggle" the callbacks and create a third, mutually incompatible callback type to handle the final result.
>
> This paper describes the mechanism used by the Twisted framework for waiting for the results of long-running operations. This mechanism, the `Deferred` , handles the often-neglected problems of error handling, callback juggling, inter-system communication and code readability.

Applications of the Twisted Framework

Two projects developed using the Twisted framework are described; one, Twisted.names, which is included as part of the Twisted distribution, a domain name server and client API, and one, Pynfo, which is packaged separately, a network information robot.

Twisted Conch: SSH in Python with Twisted

Conch is an implementation of the Secure Shell Protocol (currently in the IETF standarization process). Secure Shell (or SSH) is a popular protocol for remote shell access, file management and port forwarding protected by military-grade security. SSH supports multiple encryption and compression protocols for the wire transports, and a flexible system of multiplexed channels on top. Conch uses the Twisted networking framework to supply a library which can be used to implement both SSH clients and servers. In addition, it also contains several ready made client programs, including a drop-in replacement for the OpenSSH program from the OpenBSD project.

The Lore Document Generation Framework

Lore is a documentation generation system which uses a limited subset of XHTML, together with some class attributes, as its source format. This allows for lower barrier of entry than many other similar systems, since HTML authoring tools are plentiful as is knowledge of HTML writing. As an added advantage, the source format is viewable directly, so that even if Lore is not available the documentation is useful. It currently outputs LaTeX and HTML, which allows for most use-cases.

Perspective Broker: "Translucent" Remote Method calls in Twisted

One of the core services provided by the Twisted networking framework is "Perspective Broker" , which provides a clean, secure, easy-to-use Remote Procedure Call (RPC) mechanism. This paper explains the novel features of PB, describes the security model and its implementation, and provides brief examples of usage.

Managing the Release of a Large Python Project

Twisted is a Python networking framework. At last count, the project contains nearly 60,000 lines of effective code (not comments or blank lines). When preparing a release, many details must be checked, and many steps must be followed. We describe here the technologies and tools we use, and explain how we built tools on top of them which help us make releasing as painless as possible.

Twisted Reality: A Flexible Framework for Virtual Worlds

Flexibly modelling virtual worlds in object-oriented languages has historically been difficult; the issues arising from multiple inheritance and order-of-execution resolution have limited the sophistication of existing object-oriented simulations. Twisted Reality avoids these problems by reifying both actions and relationships, and avoiding inheritance in favor of automated composition through adapters and interfaces.

Previously

- The paper Glyph and Moshe presented in IPC10
- The errata published in IPC10 against the paper.
- A paper Moshe wrote about Twisted and Debian.

Index

P

R