The Twisted Documentation

The Twisted Development Team

June 4, 2012

Contents

Chapter 1

Introduction

1.1 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.

Chapter 2

Getting Started

2.1 Writing Servers

2.1.1 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* (page 125) 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* (page 159) for more information, or `twisted.internet.interfaces.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.

2.1.2 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; 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 set up 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* (page 152) 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.

Because this is a short example, nothing else has yet started up the Twisted reactor. `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* (page 159).

Helper Protocols

Many protocols build upon similar lower-level abstraction. The most popular in internet protocols is being line-based. Lines are usually terminated with a CR-LF combinations.

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 self.answers.has_key(line):
            self.sendLine(self.answers[line])
        else:
            self.sendLine(self.answers[None])
```

Note that the delimiter is not part of the line.

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

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.

2.1.3 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()
```

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()
```

2.1.4 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.

```
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.users.has_key(self.name):
            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 self.users.has_key(name):
            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()
```

Source listing — *chat.py*

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* (page 159) wraps for you.

Here's a sample transcript of a chat session (this is text 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
```

2.2 Writing Clients

2.2.1 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* (page 125).

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.

2.2.2 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`.

2.2.3 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.

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

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

class GreeterFactory(Factory):
    def buildProtocol(self, addr):
        return Greeter()

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 = point.connect(GreeterFactory())
d.addCallback(gotProtocol)
reactor.run()
```

Regardless of the type of client endpoint, the way to set up a new connection is simply to call the `connect` method on it and pass in a factory. 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 before calling `connect`.

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* (page 159).

> **Note:**If you've used `ClientFactory` before, make sure you remember 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.

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.

2.2.4 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 `client ConnectionLost` is called when a connection was made and then disconnected.

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)
```

2.2.5 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.

```
# 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 succesfully 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()
```

Source listing — *ircLogBot.py*

`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` that 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
```

2.2.6 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* (page 161) document for more information on using interfaces in Twisted.

2.3 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[1] 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[2]). 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[3] of the Python Library Reference[4]. There is also a ton of information available online and in books.

2.3.1 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 a 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* (page 27).

> **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.

2.3.2 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`:

```
# -*- 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
```

Source listing — *base_1.py*

(Ignore the `test-case-name` comment for now. You'll see why that's useful *below* (page 23).)

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:

[1] http://en.wikipedia.org/wiki/Unit_test
[2] http://en.wikipedia.org/wiki/Test-driven_development
[3] http://docs.python.org/library/unittest.html
[4] http://docs.python.org/library/index.html

```
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)
```

Source listing — *test_base_1.py*

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, there are two things you must get set up. Make sure you get these both done - nothing below will work unless you do.

First, make sure that the directory that *contains* your `calculus` directory is in your Python load path. If you're using the Bash shell on some form of unix (e.g., Linux, Mac OS X), run `PYTHONPATH="$PYTHONPATH:'pwd'/."` at the command line in the `calculus` directory. Once you have your Python path set up correctly, you should be able to run Python from the command line and `import calculus` without seeing an import error.

Second, make sure you can run the `trial` command. That is, make sure the directory containing the `trial` program on you system is in your shell's `PATH`. The easiest way to check if you have this is to try running `trial --help` at the command line. If you see a list of invocation options, you're in business. If your shell reports something like `trial: command not found`, make sure you have Twisted installed properly, and that the Twisted `bin` directory is in your `PATH`. If you don't know how to do this, get some local help, or figure it out by searching online for information on setting and changing environment variables for you operating system.

With those (one-time) preliminary steps out of the way, let's perform the tests. Run `trial calculus.test.test_base_1` from the command line from the `calculus` directory. You should see the following output (though your files are probably not in `/tmp`:

```
$ 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 `trial calculus.test.test_base_1.CalculationTestCase.test_subtract` from the shell.

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

The tests can be run by `trial` in multiple ways:

- `trial calculus`: run all the tests for the calculus package.
- `trial calculus.test`: run using Python's `import` notation.
- `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.
- `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.
- `trial calculus/test`: run all the tests in the test directory (not recommended).
- `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 create 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.

2.3.3 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`:

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

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
```

Source listing — *base_2.py*

We'll also create a new version of test_base_1 which imports and tests this new implementation, in `calculus/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)
```

test_base_2 — *test_base_2.py*

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

```
$ 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 tests 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:

```
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)
```

Source listing — *test_base_2b.py*

Much cleaner, no?

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`:

```
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)
```

Source listing — *test_base_3.py*

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 mutiply 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`:

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

class Calculation(object):
    def _make_ints(self, *args):
        try:
            return map(int, args)
        except ValueError:
            raise TypeError("Couldn't coerce arguments to integers: %s" % 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
```

Source listing — *base_3.py*

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.

2.3.4 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* (page 8), *Writing Clients* (page 13), and *Deferreds* (page 135).

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

2.3.5 Testing a protocol

We'll now create a custom protocol to invoke our class from within 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:

```
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('%s %d %d\r\n' % (operation, a, b))
        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)
```

Source listing — *test_remote_1.py*

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 the work off 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`:

```
# -*- 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_%s' % m, getattr(self.calc, m))



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


    def lineReceived(self, line):
        op, a, b = line.split()
        a = int(a)
        b = int(b)
        op = getattr(self.proxy, 'remote_%s' % (op,))
```

```
        result = op(a, b)
        self.sendLine(str(result))



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()
```

Source listing — *remote_1.py*

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 (`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 a
```

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 particulary useful for now : we'll now build a client to 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`:

```
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(), '%s %d %d\r\n' % (operation, a, b))
        self.tr.clear()
        d.addCallback(self.assertEqual, expected)
        self.proto.dataReceived("%d\r\n" % (expected,))
        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)
```

Source listing — *test_client_1.py*

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. 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`:

```
# -*- 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 = "%s %d %d" % (op, a, b)
        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)
```

Source listing — *client_1.py*

2.3.6 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.

```
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(), '%s %d %d\r\n' % (operation, a, b))
```

```
        self.tr.clear()
        d.addCallback(self.assertEqual, expected)
        self.proto.dataReceived("%d\r\n" % (expected,))
        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(), 'add 9 4\r\n')
        self.clock.advance(self.proto.timeOut)
        return self.assertFailure(d, ClientTimeoutError)
```

Source listing — *test_client_2.py*

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 calls in the protocol will finish before `clock.advance()` returns.

In the new test (test_timeout), we call `clock.advance`, that simulates and 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.

```
# -*- 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 = "%s %d %d" % (op, a, b)
        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)
```

Source listing — *client_2.py*

The only important thing here is to not forget to cancel our callLater when everything went fine.

Cleaning up after tests

This chapter is mainly intended for people that want to have sockets or processes created in their tests. If it's still not obvious, you must try to avoid that like the plague, 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.

```
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)
```

Source listing — *test_remote_2.py*

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

Also, you should be aware that `tearDown` will called in any case, after success or failure. So don't expect that every objects you created in the test method are 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 remove 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 he sends 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")
```

```
# -*- 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_%s' % m, getattr(self.calc, m))



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


    def lineReceived(self, line):
        op, a, b = line.split()
        op = getattr(self.proxy, 'remote_%s' % (op,))
        try:
            result = op(a, b)
        except TypeError:
            log.err()
            self.sendLine("error")
        else:
            self.sendLine(str(result))



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()
```

Source listing — *remote_2.py*

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

```
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:

```
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('%s %d %d\r\n' % (operation, a, b))
        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('add foo bar\r\n')
        self.assertEqual(self.tr.value(), "error\r\n")
        errors = self.flushLoggedErrors(TypeError)
        self.assertEqual(len(errors), 1)
```

Source listing — *test_remote_3.py*

2.3.7 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 an user will come to you saying "I have this strange problem on my crappy network environment. 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.

```
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(), '%s %d %d\r\n' % (operation, a, b))
        self.tr.clear()
        d.addCallback(self.assertEqual, expected)
        self.proto.dataReceived("%d\r\n" % (expected,))
        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(), '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(), '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)
```

Source listing — *test_client_3.py*

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.

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

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



class ClientTimeoutError(Exception):
```

```
    pass


class RemoteCalculationClient(object, 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 = "%s %d %d" % (op, a, b)
        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)
```

Source listing — *client_3.py*

2.3.8 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 kind 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 modules used during the tests. The ones interesting for us are calculus.base.cover and calculus.remote.cover. In front of each line is the number of times you went through during

the tests, or the marker '>>>>>>' if the line was not covered. If you went through all the 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 possibile input and output, to get *full* coverage (condition, path, etc.) as well .

2.3.9 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 on the Twisted web site[5].

2.4 Twisted from Scratch, or The Evolution of Finger

2.4.1 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* (page **??**) 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* (page **??**) 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.

2.4.2 Contents

This tutorial is split into eleven parts:

1. *The Evolution of Finger: building a simple finger service* (page 41)
2. *The Evolution of Finger: adding features to the finger service* (page 48)
3. *The Evolution of Finger: cleaning up the finger code* (page 58)
4. *The Evolution of Finger: moving to a component based architecture* (page 61)
5. *The Evolution of Finger: pluggable backends* (page 71)
6. *The Evolution of Finger: a web frontend* (page 83)
7. *The Evolution of Finger: Twisted client support using Perspective Broker* (page 89)
8. *The Evolution of Finger: using a single factory for multiple protocols* (page 95)
9. *The Evolution of Finger: a Twisted finger client* (page 101)
10. *The Evolution of Finger: making a finger library* (page 104)
11. *The Evolution of Finger: configuration and packaging of the finger service* (page 106)

[5] http://twistedmatrix.com/

2.5 The Evolution of Finger: building a simple finger service

2.5.1 Introduction

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

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* (page 259) 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* (page 8) and deferreds as introduced in *Using Deferreds* (page 135) and *Generating Deferreds* (page 145). Services and applications are discussed in *Using the Twisted Application Framework* (page 178).

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.

2.5.2 Refuse Connections

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

Source listing — *finger01.py*

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* (page 119) for more information.

2.5.3 Do Nothing

```
from twisted.internet import protocol, reactor

class FingerProtocol(protocol.Protocol):
    pass
```

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

reactor.listenTCP(1079, FingerFactory())
reactor.run()
```

Source listing — *finger02.py*

Here, `reactor.listenTCP` opens port 1079. (The number 1079 is a reminder that eventually we want to run on port 79, the standard port for finger servers.) The specified factory, `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.

2.5.4 Drop Connections

```
from twisted.internet import protocol, reactor

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

class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol

reactor.listenTCP(1079, FingerFactory())
reactor.run()
```

Source listing — *finger03.py*

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.

2.5.5 Read Username, Drop Connections

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

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

class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol

reactor.listenTCP(1079, FingerFactory())
reactor.run()
```

Source listing — *finger04.py*

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.

2.5.6 Read Username, Output Error, Drop Connections

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

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

class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol

reactor.listenTCP(1079, FingerFactory())
reactor.run()
```

Source listing — *finger05.py*

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.

2.5.7 Output From Empty Factory

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

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

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

class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol

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

reactor.listenTCP(1079, FingerFactory())
reactor.run()
```

Source listing — *finger06.py*

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.

2.5.8 Output from Non-empty Factory

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

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

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

class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol

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

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

reactor.listenTCP(1079, FingerFactory(moshez='Happy and well'))
reactor.run()
```

Source listing — *finger07.py*

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.

2.5.9 Use Deferreds

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

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 + '\r\n')
            self.transport.loseConnection()
        d.addCallback(writeResponse)

class FingerFactory(protocol.ServerFactory):
```

```
    protocol = FingerProtocol

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

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

reactor.listenTCP(1079, FingerFactory(moshez='Happy and well'))
reactor.run()
```

Source listing — *finger08.py*

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* (page 135), 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.

2.5.10 Run 'finger' Locally

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

from twisted.internet import protocol, reactor, defer, utils
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 + '\r\n')
            self.transport.loseConnection()
        d.addCallback(writeResponse)

class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol

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

reactor.listenTCP(1079, FingerFactory())
reactor.run()
```

Source listing — *finger09.py*

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.

2.5.11 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.

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

from twisted.internet import protocol, reactor, defer, utils
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 'Internal error in server'
        d.addErrback(onError)

        def writeResponse(message):
            self.transport.write(message + '\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)

reactor.listenTCP(1079, FingerFactory(prefix='http://livejournal.com/~'))
reactor.run()
```

Source listing — *finger10.py*

2.5.12 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.

2.5.13 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* (page 261) file).

```
# 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 internet, service
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 + '\r\n')
            self.transport.loseConnection()
        d.addCallback(writeResponse)

class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol

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

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

```
application = service.Application('finger', uid=1, gid=1)
factory = FingerFactory(moshez='Happy and well')
internet.TCPServer(79, factory).setServiceParent(
    service.IServiceCollection(application))
```

Source listing — *finger11.tac*

Instead of using `reactor.listenTCP` as in the above examples, here we are using its application-aware counterpart, `internet.TCPServer`. Notice that when it is instantiated, the application object itself does not reference either the protocol or the factory. Any services (such as TCPServer) 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 TCPServer we opened, the application lets us manage the TCPServer.

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

2.6 The Evolution of Finger: adding features to the finger service

2.6.1 Introduction

This is the second part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 40).

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* (page **??**). More examples using `twisted.words.protocols.irc` can be found in *Writing a TCP Client* (page 13) and the *Twisted Words examples* (page **??**).

2.6.2 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

# But let's try and fix setting away messages, shall we?
from twisted.application import internet, service
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 + '\r\n')
            self.transport.loseConnection()
        d.addCallback(writeResponse)
```

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

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

    def getUser(self, user):
        return defer.succeed(self.users.get(user, "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(moshez='Happy and well')
fsf = FingerSetterFactory(ff)

application = service.Application('finger', uid=1, gid=1)
serviceCollection = service.IServiceCollection(application)
internet.TCPServer(79,ff).setServiceParent(serviceCollection)
internet.TCPServer(1079,fsf).setServiceParent(serviceCollection)
```

Source listing — *finger12.tac*

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.

2.6.3 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.

```
# Fix asymmetry
from twisted.application import internet, service
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 + '\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, **kwargs):
        self.users = kwargs

    def getUser(self, user):
        return defer.succeed(self.users.get(user, "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(moshez='Happy and well')
serviceCollection = service.IServiceCollection(application)
internet.TCPServer(79,f.getFingerFactory()
                   ).setServiceParent(serviceCollection)
internet.TCPServer(1079,f.getFingerSetterFactory()
                   ).setServiceParent(serviceCollection)
```

Source listing — *finger13.tac*

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

2.6.4 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.

```
moshez: happy and well
shawn: alive
```

sample /etc/users file — *etc.users*

```
# Read from file
from twisted.application import internet, service
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 + '\r\n')
            self.transport.loseConnection()
        d.addCallback(writeResponse)


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

    def _read(self):
        for line in file(self.filename):
            user, status = line.split(':', 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, "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 = internet.TCPServer(79, f.getFingerFactory())

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

Source listing — *finger14.tac*

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.

2.6.5 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.

```
# Read from file, announce on the web!
from twisted.application import internet, service
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 'Internal error in server'
        d.addErrback(onError)

        def writeResponse(message):
            self.transport.write(message + '\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 a string.
        'request' is a 'twisted.web.server.Request'.
        """
        messagevalue = self.users.get(username)
        username = cgi.escape(username)
        if messagevalue is not None:
            messagevalue = cgi.escape(messagevalue)
            text = '<h1>%s</h1><p>%s</p>' % (username,messagevalue)
        else:
            text = '<h1>%s</h1><p>No such user</p>' % username
        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()
        for line in file(self.filename):
            user, status = line.split(':', 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, "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)
internet.TCPServer(79, f.getFingerFactory()
                   ).setServiceParent(serviceCollection)
internet.TCPServer(8000, server.Site(f.getResource())
                   ).setServiceParent(serviceCollection)
```

Source listing — *finger15.tac*

2.6.6 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 reconnecting client factory 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`.

```
# Read from file, announce on the web, irc
from twisted.application import internet, service
from twisted.internet import protocol, reactor, defer
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 'Internal error in server'
        d.addErrback(onError)

        def writeResponse(message):
            self.transport.write(message + '\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)

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

            def writeResponse(message):
                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()
        for line in file(self.filename):
            user, status = line.split(':', 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, "No such user"))

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

    def getResource(self):
        r = resource.Resource()
        r.getChild = (lambda path, request:
                      static.Data('<h1>%s</h1><p>%s</p>' %
                      tuple(map(cgi.escape,
                      [path,self.users.get(path,
                      "No such user <p/> usage: site/user")])),
                      'text/html'))
        return r

    def getIRCBot(self, nickname):
        f = protocol.ReconnectingClientFactory()
        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)
internet.TCPServer(79, f.getFingerFactory()
                   ).setServiceParent(serviceCollection)
internet.TCPServer(8000, server.Site(f.getResource())
                   ).setServiceParent(serviceCollection)
internet.TCPClient('irc.freenode.org', 6667, f.getIRCBot('fingerbot')
                   ).setServiceParent(serviceCollection)
```

Source listing — *finger16.tac*

`FingerService` now has another new function, `getIRCbot`, which returns the `ReconnectingClient Factory`. 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.

2.6.7 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.

```
# Read from file, announce on the web, irc, xml-rpc
from twisted.application import internet, service
from twisted.internet import protocol, reactor, defer
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 'Internal error in server'
        d.addErrback(onError)

        def writeResponse(message):
            self.transport.write(message + '\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)

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

            def writeResponse(message):
                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()
        for line in file(self.filename):
            user, status = line.split(':', 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, "No such user"))

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

    def getResource(self):
        r = resource.Resource()
        r.getChild = (lambda path, request:
                      static.Data('<h1>%s</h1><p>%s</p>' %
                      tuple(map(cgi.escape,
                      [path,self.users.get(path, "No such user")])),
                      'text/html'))
        x = xmlrpc.XMLRPC()
        x.xmlrpc_getUser = self.getUser
        r.putChild('RPC2', x)
        return r

    def getIRCBot(self, nickname):
        f = protocol.ReconnectingClientFactory()
        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)
internet.TCPServer(79, f.getFingerFactory()
                   ).setServiceParent(serviceCollection)
internet.TCPServer(8000, server.Site(f.getResource())
                   ).setServiceParent(serviceCollection)
internet.TCPClient('irc.freenode.org', 6667, f.getIRCBot('fingerbot')
                   ).setServiceParent(serviceCollection)
```

Source listing — *finger17.tac*

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`.

```
# testing xmlrpc finger

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

Source listing — *fingerXRclient.py*

2.7 The Evolution of Finger: cleaning up the finger code

2.7.1 Introduction

This is the third part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 40).

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

2.7.2 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.

```
# Do everything properly
from twisted.application import internet, service
from twisted.internet import protocol, reactor, defer
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+'\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)
            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()
        for line in file(self.filename):
            user, status = line.split(':', 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, "No such user"))

    def getUsers(self):
        return defer.succeed(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.ReconnectingClientFactory()
        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)
internet.TCPServer(79, f.getFingerFactory()
                   ).setServiceParent(serviceCollection)
internet.TCPServer(8000, server.Site(f.getResource())
                   ).setServiceParent(serviceCollection)
internet.TCPClient('irc.freenode.org', 6667, f.getIRCBot('fingerbot')
                   ).setServiceParent(serviceCollection)
```

Source listing — *finger18.tac*

2.8 The Evolution of Finger: moving to a component based architecture

2.8.1 Introduction

This is the fourth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 40).

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* (page 161) for a more complete discussion of components.

2.8.2 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 a call to `zope.interface.implements`. 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.

```
# Do everything properly, and componentize
from twisted.application import internet, service
```

```
from twisted.internet import protocol, reactor, defer
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, implements
import cgi

class IFingerService(Interface):

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

    def getUsers():
        """
        Return a deferred returning a list of strings.
        """


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.
        """
```

```
class FingerFactoryFromService(protocol.ServerFactory):

    implements(IFingerFactory)

    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.
        """


class FingerSetterFactoryFromService(protocol.ServerFactory):

    implements(IFingerSetterFactory)

    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.
        """


class IRCClientFactoryFromService(protocol.ClientFactory):

    implements(IIRCClientFactory)

    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):

    implements(resource.IResource)
```

```
    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)


class FingerService(service.Service):

    implements(IFingerService)
```

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

    def _read(self):
        self.users.clear()
        for line in file(self.filename):
            user, status = line.split(':', 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, "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()


application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
serviceCollection = service.IServiceCollection(application)
f.setServiceParent(serviceCollection)
internet.TCPServer(79, IFingerFactory(f)
                   ).setServiceParent(serviceCollection)
internet.TCPServer(8000, server.Site(resource.IResource(f))
                   ).setServiceParent(serviceCollection)
i = IIRCClientFactory(f)
i.nickname = 'fingerbot'
internet.TCPClient('irc.freenode.org', 6667, i
                   ).setServiceParent(serviceCollection)
```

Source listing — *finger19.tac*

2.8.3 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

```
class IFingerSetterService(Interface):

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

```
# Advantages of latest version

class MemoryFingerService(service.Service):

    implements([IFingerService, IFingerSetterService])

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

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

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

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


f = MemoryFingerService(moshez='Happy and well')
serviceCollection = service.IServiceCollection(application)
internet.TCPServer(1079, IFingerSetterFactory(f), interface='127.0.0.1'
                   ).setServiceParent(serviceCollection)
```

Source listing — *finger19a_changes.py*

Full source code here:

```
# Do everything properly, and componentize
from twisted.application import internet, service
from twisted.internet import protocol, reactor, defer
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, implements
import cgi

class IFingerService(Interface):

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

    def getUsers():
        """Return a deferred returning a list of strings"""

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"""


class FingerFactoryFromService(protocol.ServerFactory):

    implements(IFingerFactory)

    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"""


class FingerSetterFactoryFromService(protocol.ServerFactory):
```

```
    implements(IFingerSetterFactory)

    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"""


class IRCClientFactoryFromService(protocol.ClientFactory):

    implements(IIRCClientFactory)

    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):

    implements(resource.IResource)

    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)

class MemoryFingerService(service.Service):

    implements([IFingerService, IFingerSetterService])

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

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

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

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


application = service.Application('finger', uid=1, gid=1)
f = MemoryFingerService(moshez='Happy and well')
serviceCollection = service.IServiceCollection(application)
internet.TCPServer(79, IFingerFactory(f)
                   ).setServiceParent(serviceCollection)
internet.TCPServer(8000, server.Site(resource.IResource(f))
                   ).setServiceParent(serviceCollection)
i = IIRCClientFactory(f)
i.nickname = 'fingerbot'
internet.TCPClient('irc.freenode.org', 6667, i
                   ).setServiceParent(serviceCollection)
internet.TCPServer(1079, IFingerSetterFactory(f), interface='127.0.0.1'
                   ).setServiceParent(serviceCollection)
```

Source listing — *finger19a.tac*

2.8.4 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!

2.9 The Evolution of Finger: pluggable backends

2.9.1 Introduction

This is the fifth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 40).

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* (page 61). 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.

2.9.2 Another Back-end

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

```
# Another back-end

class LocalFingerService(service.Service):

    implements(IFingerService)

    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()
```

Source listing — *finger19b_changes.py*

Full source code here:

```
# Do everything properly, and componentize
from twisted.application import internet, service
from twisted.internet import protocol, reactor, defer, utils
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, implements
import cgi
import pwd

class IFingerService(Interface):

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

    def getUsers():
        """
        Return a deferred returning a list of strings.
        """


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+'\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.
        """


class FingerFactoryFromService(protocol.ServerFactory):

    implements(IFingerFactory)

    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.
        """


class FingerSetterFactoryFromService(protocol.ServerFactory):

    implements(IFingerSetterFactory)

    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.
        """


class IRCClientFactoryFromService(protocol.ClientFactory):

    implements(IIRCClientFactory)

    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):

    implements(resource.IResource)

    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)


class FingerService(service.Service):

    implements(IFingerService)

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

    def _read(self):
        self.users.clear()
        for line in file(self.filename):
            user, status = line.split(':', 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, "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()
```

```
# Another back-end

class LocalFingerService(service.Service):

    implements(IFingerService)

    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([])


application = service.Application('finger', uid=1, gid=1)
f = LocalFingerService()
serviceCollection = service.IServiceCollection(application)
internet.TCPServer(79, IFingerFactory(f)
                   ).setServiceParent(serviceCollection)
internet.TCPServer(8000, server.Site(resource.IResource(f))
                   ).setServiceParent(serviceCollection)
i = IIRCClientFactory(f)
i.nickname = 'fingerbot'
internet.TCPClient('irc.freenode.org', 6667, i
                   ).setServiceParent(serviceCollection)
```

Source listing — *finger19b.tac*

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

2.9.3 Yet Another Back-end: Doing the Standard Thing

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


# Yet another back-end

class LocalFingerService(service.Service):

    implements(IFingerService)

    def getUser(self, user):
        user = user.strip()
        try:
            entry = pwd.getpwnam(user)
        except KeyError:
            return defer.succeed("No such user")
        try:
            f = file(os.path.join(entry[5],'.plan'))
        except (IOError, OSError):
            return defer.succeed("No such user")
        data = f.read()
```

```
        data = data.strip()
        f.close()
        return defer.succeed(data)

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



f = LocalFingerService()
```

Source listing — *finger19c_changes.py*

Full source code here:

```
# Do everything properly, and componentize
from twisted.application import internet, service
from twisted.internet import protocol, reactor, defer, utils
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, implements
import cgi
import pwd
import os

class IFingerService(Interface):

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

    def getUsers():
        """
        Return a deferred returning a list of strings.
        """


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+'\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.
        """


class FingerFactoryFromService(protocol.ServerFactory):

    implements(IFingerFactory)

    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.
        """


class FingerSetterFactoryFromService(protocol.ServerFactory):

    implements(IFingerSetterFactory)

    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.
        """


class IRCClientFactoryFromService(protocol.ClientFactory):

    implements(IIRCClientFactory)

    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):

    implements(resource.IResource)

    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)


class FingerService(service.Service):

    implements(IFingerService)

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

    def _read(self):
        self.users.clear()
        for line in file(self.filename):
            user, status = line.split(':', 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, "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()


# Yet another back-end
```

```
class LocalFingerService(service.Service):

    implements(IFingerService)

    def getUser(self, user):
        user = user.strip()
        try:
            entry = pwd.getpwnam(user)
        except KeyError:
            return defer.succeed("No such user")
        try:
            f = file(os.path.join(entry[5],'.plan'))
        except (IOError, OSError):
            return defer.succeed("No such user")
        data = f.read()
        data = data.strip()
        f.close()
        return defer.succeed(data)

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


application = service.Application('finger', uid=1, gid=1)
f = LocalFingerService()
serviceCollection = service.IServiceCollection(application)
internet.TCPServer(79, IFingerFactory(f)
                   ).setServiceParent(serviceCollection)
internet.TCPServer(8000, server.Site(resource.IResource(f))
                   ).setServiceParent(serviceCollection)
i = IIRCClientFactory(f)
i.nickname = 'fingerbot'
internet.TCPClient('irc.freenode.org', 6667, i
                   ).setServiceParent(serviceCollection)
```

Source listing — *finger19c.tac*

Not much to say except that now we can be churn out backends like crazy. Feel like doing a back-end for Advogato[6], for example? Dig out the XML-RPC client support Twisted has, and get to work!

2.10 The Evolution of Finger: a web frontend

2.10.1 Introduction

This is the sixth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 40).

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.

```
# Do everything properly, and componentize
from twisted.application import internet, service
```

[6] http://www.advogato.org/

```
from twisted.internet import protocol, reactor, defer
from twisted.words.protocols import irc
from twisted.protocols import basic
from twisted.python import components
from twisted.web import resource, server, static, xmlrpc, microdom
from zope.interface import Interface, implements
import cgi

class IFingerService(Interface):

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

    def getUsers():
        """
        Return a deferred returning a list of strings.
        """


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.
        """
```

```
class FingerFactoryFromService(protocol.ServerFactory):

    implements(IFingerFactory)

    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.
        """


class FingerSetterFactoryFromService(protocol.ServerFactory):

    implements(IFingerSetterFactory)

    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.
        """


class IRCClientFactoryFromService(protocol.ClientFactory):

    implements(IIRCClientFactory)

    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 FingerService(service.Service):

    implements(IFingerService)

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

    def _read(self):
        self.users.clear()
        for line in file(self.filename):
            user, status = line.split(':', 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, "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()


application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
serviceCollection = service.IServiceCollection(application)
f.setServiceParent(serviceCollection)
internet.TCPServer(79, IFingerFactory(f)
                   ).setServiceParent(serviceCollection)
internet.TCPServer(8000, server.Site(resource.IResource(f))
                   ).setServiceParent(serviceCollection)
i = IIRCClientFactory(f)
i.nickname = 'fingerbot'
internet.TCPClient('irc.freenode.org', 6667, i
                   ).setServiceParent(serviceCollection)
```

Source listing — *finger20.tac*

2.11 The Evolution of Finger: Twisted client support using Perspective Broker

2.11.1 Introduction

This is the seventh part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 40).

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* (page **??**) of the core howto index.

2.11.2 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.

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

class IFingerService(Interface):

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

    def getUsers():
        """
        Return a deferred returning a list of strings.
        """


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.
        """


class FingerFactoryFromService(protocol.ServerFactory):

    implements(IFingerFactory)

    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.
        """


class FingerSetterFactoryFromService(protocol.ServerFactory):

    implements(IFingerSetterFactory)

    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.
        """
```

```
class IRCClientFactoryFromService(protocol.ClientFactory):

    implements(IIRCClientFactory)

    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.
        """


class PerspectiveFingerFromService(pb.Root):

    implements(IPerspectiveFinger)

    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)


class FingerService(service.Service):

    implements(IFingerService)

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

    def _read(self):
        self.users.clear()
        for line in file(self.filename):
            user, status = line.split(':', 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, "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()


application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
serviceCollection = service.IServiceCollection(application)
f.setServiceParent(serviceCollection)
internet.TCPServer(79, IFingerFactory(f)
                   ).setServiceParent(serviceCollection)
internet.TCPServer(8000, server.Site(resource.IResource(f))
                   ).setServiceParent(serviceCollection)
i = IIRCClientFactory(f)
i.nickname = 'fingerbot'
internet.TCPClient('irc.freenode.org', 6667, i
                   ).setServiceParent(serviceCollection)
internet.TCPServer(8889, pb.PBServerFactory(IPerspectiveFinger(f))
                   ).setServiceParent(serviceCollection)
```

Source listing — *finger21.tac*

A simple client to test the perspective broker finger:

```
# test the PB finger on port 8889
# this code is essentially the same as
# the first example in howto/pb-usage

from twisted.spread import pb
from twisted.internet import reactor

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()
reactor.connectTCP("127.0.0.1",8889, factory)
factory.getRootObject().addCallbacks(gotObject,gotNoObject)
reactor.run()
```

Source listing — *fingerPBclient.py*

2.12 The Evolution of Finger: using a single factory for multiple protocols

2.12.1 Introduction

This is the eighth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 40).

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* (page 120).

2.12.2 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.application.internet.SSLServer method. Note that one factory (in this case, a site) can listen on multiple ports with multiple protocols.

```
# Do everything properly, and componentize
from twisted.application import internet, service
from twisted.internet import protocol, reactor, defer
from twisted.words.protocols import irc
from twisted.protocols import basic
from twisted.python import components
from twisted.web import resource, server, static, xmlrpc, microdom
from twisted.spread import pb
from zope.interface import Interface, implements
from OpenSSL import SSL
import cgi

class IFingerService(Interface):
```

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

    def getUsers():
        """
        Return a deferred returning a list of strings.
        """


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.
        """


class FingerFactoryFromService(protocol.ServerFactory):

    implements(IFingerFactory)

    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.
        """


class FingerSetterFactoryFromService(protocol.ServerFactory):

    implements(IFingerSetterFactory)

    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.
        """


class IRCClientFactoryFromService(protocol.ClientFactory):

    implements(IIRCClientFactory)

    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.
        """

class PerspectiveFingerFromService(pb.Root):

    implements(IPerspectiveFinger)

    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)


class FingerService(service.Service):

    implements(IFingerService)

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

    def _read(self):
        self.users.clear()
        for line in file(self.filename):
            user, status = line.split(':', 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, "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()


class ServerContextFactory:

    def getContext(self):
        """
        Create an SSL context.

        This is a sample implementation that loads a certificate from a file
        called 'server.pem'.
        """
        ctx = SSL.Context(SSL.SSLv23_METHOD)
        ctx.use_certificate_file('server.pem')
        ctx.use_privatekey_file('server.pem')
        return ctx


application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
serviceCollection = service.IServiceCollection(application)
f.setServiceParent(serviceCollection)
internet.TCPServer(79, IFingerFactory(f)
                   ).setServiceParent(serviceCollection)
site = server.Site(resource.IResource(f))
internet.TCPServer(8000, site
                   ).setServiceParent(serviceCollection)
internet.SSLServer(443, site, ServerContextFactory()
                   ).setServiceParent(serviceCollection)
i = IIRCClientFactory(f)
i.nickname = 'fingerbot'
internet.TCPClient('irc.freenode.org', 6667, i
                   ).setServiceParent(serviceCollection)
internet.TCPServer(8889, pb.PBServerFactory(IPerspectiveFinger(f))
                   ).setServiceParent(serviceCollection)
```

Source listing — *finger22.py*

2.13 The Evolution of Finger: a Twisted finger client

2.13.1 Introduction

This is the ninth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 40).

In this part, we develop a client for the finger server: a proxy finger server which forwards requests to another finger server.

2.13.2 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.

```
# finger proxy
from twisted.application import internet, service
from twisted.internet import defer, protocol, reactor
from twisted.protocols import basic
from twisted.python import components
from zope.interface import Interface, implements


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

class IFingerService(Interface):

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

    def getUsers():
        """Return a deferred returning a list of strings"""


class IFingerFactory(Interface):

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

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

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)



class FingerFactoryFromService(protocol.ClientFactory):

    implements(IFingerFactory)

    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+"\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)
    reactor.connectTCP(host, port, f)
    return f.d


class ProxyFingerService(service.Service):
    implements(IFingerService)

    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()
internet.TCPServer(7779, IFingerFactory(f)).setServiceParent(
    service.IServiceCollection(application))
```

Source listing — *fingerproxy.tac*

2.14 The Evolution of Finger: making a finger library

2.14.1 Introduction

This is the tenth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 40).

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* (page 178).

2.14.2 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
# eg:  twistd -ny organized-finger.tac

import finger

from twisted.internet import protocol, reactor, defer
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)
internet.TCPServer(79, finger.IFingerFactory(f)
                   ).setServiceParent(serviceCollection)

site = server.Site(resource.IResource(f))
internet.TCPServer(8000, site
                   ).setServiceParent(serviceCollection)

internet.SSLServer(443, site, finger.ServerContextFactory()
                   ).setServiceParent(serviceCollection)

i = finger.IIRCClientFactory(f)
i.nickname = 'fingerbot'
internet.TCPClient('irc.freenode.org', 6667, i
                   ).setServiceParent(serviceCollection)

internet.TCPServer(8889, pb.PBServerFactory(finger.IPerspectiveFinger(f))
                   ).setServiceParent(serviceCollection)
```

Source listing — *organized-finger.tac*

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.

2.14.3 Easy Configuration

We can also supply easy configuration for common cases with a makeService method that will also help build .tap files later:

```
# Easy configuration
# makeService from finger module

def makeService(config):
    # finger on port 79
    s = service.MultiService()
    f = FingerService(config['file'])
    h = internet.TCPServer(79, IFingerFactory(f))
    h.setServiceParent(s)

    # website on port 8000
    r = resource.IResource(f)
    r.templateDirectory = config['templates']
    site = server.Site(r)
    j = internet.TCPServer(8000, site)
    j.setServiceParent(s)

    # ssl on port 443
    if config.get('ssl'):
        k = internet.SSLServer(443, site, ServerContextFactory())
        k.setServiceParent(s)

    # irc fingerbot
    if config.has_key('ircnick'):
        i = IIRCClientFactory(f)
        i.nickname = config['ircnick']
        ircserver = config['ircserver']
        b = internet.TCPClient(ircserver, 6667, i)
        b.setServiceParent(s)

    # Pespective Broker on port 8889
    if config.has_key('pbport'):
        m = internet.TCPServer(
            int(config['pbport']),
            pb.PBServerFactory(IPerspectiveFinger(f)))
        m.setServiceParent(s)

    return s
```

Source listing — *finger_config.py*

And we can write simpler files now:

```
# 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))
```

Source listing — *simple-finger.tac*

```
% twisted -ny simple-finger.tac
```

Note: the finger *user* still has ultimate power: he can use `makeService`, or he can use the lower-level interface if he has 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.

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.

2.15 The Evolution of Finger: configuration and packaging of the finger service

2.15.1 Introduction

This is the eleventh part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 40).

In this part, we make it easier for non-programmers to configure a finger server and show how to package it in the .deb and RPM package formats. Plugins are discussed further in the *Twisted Plugin System* (page 173) howto. Writing twistd plugins is covered in *Writing a twistd Plugin* (page 181), and .tac applications are covered in *Using the Twisted Application Framework* (page 178).

2.15.2 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 module

from zope.interface import Interface, implements

from twisted.application import internet, service
from twisted.internet import protocol, reactor, defer
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

from OpenSSL import SSL

class IFingerService(Interface):

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

    def getUsers():
        """
        Return a deferred returning a list of strings.
        """


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+'\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.
        """


class FingerFactoryFromService(protocol.ServerFactory):
    implements(IFingerFactory)

    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.
        """


class FingerSetterFactoryFromService(protocol.ServerFactory):

    implements(IFingerSetterFactory)

    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.
        """


class IRCClientFactoryFromService(protocol.ClientFactory):

    implements(IIRCClientFactory)

    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.
        """


class PerspectiveFingerFromService(pb.Root):

    implements(IPerspectiveFinger)

    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)


class FingerService(service.Service):
```

```
    implements(IFingerService)

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

    def _read(self):
        self.users = {}
        for line in file(self.filename):
            user, status = line.split(':', 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, "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()


class ServerContextFactory:

    def getContext(self):
        """
        Create an SSL context.

        This is a sample implementation that loads a certificate from a file
        called 'server.pem'.
        """
        ctx = SSL.Context(SSL.SSLv23_METHOD)
        ctx.use_certificate_file('server.pem')
        ctx.use_privatekey_file('server.pem')
        return ctx



# Easy configuration

def makeService(config):
    # finger on port 79
    s = service.MultiService()
    f = FingerService(config['file'])
    h = internet.TCPServer(1079, IFingerFactory(f))
    h.setServiceParent(s)


    # website on port 8000
    r = resource.IResource(f)
```

```
    r.templateDirectory = config['templates']
    site = server.Site(r)
    j = internet.TCPServer(8000, site)
    j.setServiceParent(s)

    # ssl on port 443
#    if config.get('ssl'):
#        k = internet.SSLServer(443, site, ServerContextFactory())
#        k.setServiceParent(s)

    # irc fingerbot
    if config.has_key('ircnick'):
        i = IIRCClientFactory(f)
        i.nickname = config['ircnick']
        ircserver = config['ircserver']
        b = internet.TCPClient(ircserver, 6667, i)
        b.setServiceParent(s)

    # Pespective Broker on port 8889
    if config.has_key('pbport'):
        m = internet.TCPServer(
            int(config['pbport']),
            pb.PBServerFactory(IPerspectiveFinger(f)))
        m.setServiceParent(s)

    return s
```

finger module — *finger.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)
```

finger/tap.py — *tap.py*

And register it all:

```
from twisted.application.service import ServiceMaker

finger = ServiceMaker(
```

```
    'finger', 'finger.tap', 'Run a finger service', 'finger')
```

twisted/plugins/finger_tutorial.py — *finger_tutorial.py*

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* (page 181).

2.15.3 OS Integration

If we already have the "finger" package installed in `PYTHONPATH` (e.g. we added it to `site-packages`), we can achieve easy integration:

Debian

```
% tap2deb --unsigned -m "Foo <foo@example.com>" --type=python finger.tac
% sudo dpkg -i .build/*.deb
```

Red Hat / Mandrake

```
% tap2rpm --type=python finger.tac
% sudo rpm -i *.rpm
```

These packages will properly install and register `init.d` scripts, etc. for the given file.

If it doesn't work on your favorite OS: patches accepted!

2.16 Setting up the TwistedQuotes application

2.16.1 Goal

This document describes how to set up the TwistedQuotes application used in a number of other documents, such as *designing Twisted applications* (page 116).

2.16.2 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:
   - ```
     """
     Twisted Quotes
     """
     ```

     Source listing — *__init__.py*

     (this file marks it as a package, see this section[7] of the Python tutorial for more on packages)
   - ```
     from random import choice

     from zope.interface import implements

     from TwistedQuotes import quoteproto
     ```

[7] http://docs.python.org/tutorial/modules.html#packages

```
class StaticQuoter:
    """
    Return a static quote.
    """

    implements(quoteproto.IQuoter)

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


    def getQuote(self):
        return self.quote



class FortuneQuoter:
    """
    Load quotes from a fortune-format file.
    """
    implements(quoteproto.IQuoter)

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


    def getQuote(self):
        quoteFile = file(choice(self.filenames))
        quotes = quoteFile.read().split('\n%\n')
        quoteFile.close()
        return choice(quotes)
```

Source listing — *quoters.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
```

Source listing — *quoteproto.py*

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.
```

2.17 Designing Twisted Applications

2.17.1 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* (page 8) and *clients* (page 13) using Twisted.

2.17.2 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* (page 114).

A Look at the Heart of the Application

```
from random import choice

from zope.interface import implements

from TwistedQuotes import quoteproto
```

```
class StaticQuoter:
    """
    Return a static quote.
    """

    implements(quoteproto.IQuoter)

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


    def getQuote(self):
        return self.quote



class FortuneQuoter:
    """
    Load quotes from a fortune-format file.
    """
    implements(quoteproto.IQuoter)

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


    def getQuote(self):
        quoteFile = file(choice(self.filenames))
        quotes = quoteFile.read().split('\n%\n')
        quoteFile.close()
        return choice(quotes)
```

Twisted Quotes Central Abstraction — *quoters.py*

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* (page 135).

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.

```
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
```

Twisted Quotes Protocol Implementation — *quoteproto.py*

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* (page 8) 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* (page 178).

Chapter 3

Networking and Other Event Sources

3.1 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.

3.2 Reactor Overview

This HOWTO introduces the Twisted reactor, describes the basics of the reactor and links to the various reactor interfaces.

3.2.1 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* (page 149) and *Using Deferreds* (page 135);
- the communication howtos: *TCP servers* (page 8), *TCP clients* (page 13), *UDP networking* (page 125) and *Using processes* (page 128); and
- *Using threads* (page 150).

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* (page 154).

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* (page 178) for an introduction to Application.

3.2.2 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* (page 128) document for more information.
- `IReactorSSL`: SSL networking support.
- `IReactorTCP`: TCP networking support. More information can be found in the *Writing Servers* (page 8) and *Writing Clients* (page 13) documents.
- `IReactorThreads`: Threading use and management. More information can be found within *Threading In Twisted* (page 150).
- `IReactorTime`: Scheduling interface. More information can be found within *Scheduling Tasks* (page 149).
- `IReactorUDP`: UDP networking support. More information can be found within *UDP Networking* (page 125).
- `IReactorUNIX`: UNIX socket support.
- `IReactorSocket`: Third-party socket support.

3.3 Using SSL in Twisted

3.3.1 Overview

This document describes how to use SSL in Twisted servers and clients. It assumes that you know what SSL is, what some of the major reasons to use it are, and how to generate your own SSL certificates, in particular self-signed certificates. It also assumes that you are comfortable with creating TCP servers and clients as described in the *server howto* (page 8) and *client howto* (page 13). After reading this document you should be able to create servers and clients that can use SSL to encrypt their connections, switch from using an unencrypted channel to an encrypted one mid-connection, and require client authentication.

Using SSL in Twisted requires that you have pyOpenSSL[1] 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.

SSL connections require SSL contexts. These contexts are generated by a `ContextFactory` that maintains state like the SSL method, private key file name, and certificate file name.

Instead of using listenTCP and connectTCP to create a connection, use `listenSSL` and `connectSSL` for a server and client respectively. These methods take a contextFactory as an additional argument.

The basic server context factory is `twisted.internet.ssl.ContextFactory`, and the basic client context factory is `twisted.internet.ssl.ClientContextFactory`. They can be used as-is or subclassed. `twisted.internet.ssl.DefaultOpenSSLContextFactory` is a convenience server class that subclasses `ContextFactory` and adds default parameters to the SSL handshake and connection. Another useful class is `twisted.internet.ssl.CertificateOptions`; it is a factory for SSL context objects that lets you specify many of the common verification and session options so it can do the proper pyOpenSSL initialization for you.

Those are the big immediate differences between TCP and SSL connections, so let's look at an example. In it and all subsequent examples it is assumed that keys and certificates for the server, certificate authority, and client should they exist live in a keys/ subdirectory of the directory containing the example code, and that the certificates are self-signed.

[1] http://pyopenssl.sf.net

3.3.2 SSL echo server and client without client authentication

Authentication and encryption are two separate parts of the SSL protocol. The server almost always needs a key and certificate to authenticate itself to the client but is usually configured to allow encrypted connections with unauthenticated clients who don't have certificates. This common case is demonstrated first by adding SSL support to the echo client and server in the *core examples* (page **??**).

SSL echo server

```
from twisted.internet import ssl, reactor
from twisted.internet.protocol import Factory, Protocol

class Echo(Protocol):
    def dataReceived(self, data):
        """As soon as any data is received, write it back."""
        self.transport.write(data)

if __name__ == '__main__':
    factory = Factory()
    factory.protocol = Echo
    reactor.listenSSL(8000, factory,
                      ssl.DefaultOpenSSLContextFactory(
            'keys/server.key', 'keys/server.crt'))
    reactor.run()
```

SSL echo client

```
from twisted.internet import ssl, reactor
from twisted.internet.protocol import ClientFactory, Protocol

class EchoClient(Protocol):
    def connectionMade(self):
        print "hello, world"
        self.transport.write("hello, world!")

    def dataReceived(self, data):
        print "Server said:", data
        self.transport.loseConnection()

class EchoClientFactory(ClientFactory):
    protocol = EchoClient

    def clientConnectionFailed(self, connector, reason):
        print "Connection failed - goodbye!"
        reactor.stop()

    def clientConnectionLost(self, connector, reason):
        print "Connection lost - goodbye!"
        reactor.stop()

if __name__ == '__main__':
    factory = EchoClientFactory()
    reactor.connectSSL('localhost', 8000, factory, ssl.ClientContextFactory())
    reactor.run()
```

Contexts are created according to a specified method. `SSLv3_METHOD`, `SSLv23_METHOD`, and `TLSv1_METHOD` are the valid constants that represent SSL methods to use when creating a context object. `DefaultOpenSSLContextFactory` and `ClientContextFactory` default to using `SSL.SSLv23_METHOD` as their method,

and it is compatible for communication with all the other methods listed above. An older method constant, `SSLv2_METHOD`, exists but is explicitly disallowed in both `DefaultOpenSSLContextFactory` and `ClientContextFactory` for being insecure by calling `set_options(SSL.OP_NO_SSLv2)` on their contexts. See `twisted.internet.ssl` for additional comments.

3.3.3 Using startTLS

If you want to switch from unencrypted to encrypted traffic mid-connection, you'll need to turn on SSL 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[2].

startTLS server

```
from OpenSSL import SSL
from twisted.internet import reactor, ssl
from twisted.internet.protocol import ServerFactory
from twisted.protocols.basic import LineReceiver

class TLSServer(LineReceiver):
    def lineReceived(self, line):
        print "received: " + line

        if line == "STARTTLS":
            print "-- Switching to TLS"
            self.sendLine('READY')
            ctx = ServerTLSContext(
                privateKeyFileName='keys/server.key',
                certificateFileName='keys/server.crt',
                )
            self.transport.startTLS(ctx, self.factory)


class ServerTLSContext(ssl.DefaultOpenSSLContextFactory):
    def __init__(self, *args, **kw):
        kw['sslmethod'] = SSL.TLSv1_METHOD
        ssl.DefaultOpenSSLContextFactory.__init__(self, *args, **kw)

if __name__ == '__main__':
    factory = ServerFactory()
    factory.protocol = TLSServer
    reactor.listenTCP(8000, factory)
    reactor.run()
```

startTLS client

```
from OpenSSL import SSL
from twisted.internet import reactor, ssl
from twisted.internet.protocol import ClientFactory
from twisted.protocols.basic import LineReceiver

class ClientTLSContext(ssl.ClientContextFactory):
    isClient = 1
    def getContext(self):
        return SSL.Context(SSL.TLSv1_METHOD)
```

[2] http://www.wireshark.org/

```
class TLSClient(LineReceiver):
    pretext = [
        "first line",
        "last thing before TLS starts",
        "STARTTLS"]

    posttext = [
        "first thing after TLS started",
        "last thing ever"]

    def connectionMade(self):
        for l in self.pretext:
            self.sendLine(l)

    def lineReceived(self, line):
        print "received: " + line
        if line == "READY":
            ctx = ClientTLSContext()
            self.transport.startTLS(ctx, self.factory)
            for l in self.posttext:
                self.sendLine(l)
            self.transport.loseConnection()

class TLSClientFactory(ClientFactory):
    protocol = TLSClient

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

    def clientConnectionLost(self, connector, reason):
        print "connection lost: ", reason.getErrorMessage()
        reactor.stop()

if __name__ == "__main__":
    factory = TLSClientFactory()
    reactor.connectTCP('localhost', 8000, factory)
    reactor.run()
```

`startTLS` is a transport method that gets passed a context. It is invoked at an agreed-upon time in the data reception method of the client and server protocols. The `ServerTLSContext` and `ClientTLSContext` classes used above inherit from the basic server and client context factories used in the earlier echo examples and illustrate two more ways of setting an SSL method.

3.3.4 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.

Client-authenticating server

```
from OpenSSL import SSL
from twisted.internet import ssl, reactor
from twisted.internet.protocol import Factory, Protocol

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

```
def verifyCallback(connection, x509, errnum, errdepth, ok):
    if not ok:
        print 'invalid cert from subject:', x509.get_subject()
        return False
    else:
        print "Certs are fine"
    return True

if __name__ == '__main__':
    factory = Factory()
    factory.protocol = Echo

    myContextFactory = ssl.DefaultOpenSSLContextFactory(
        'keys/server.key', 'keys/server.crt'
        )

    ctx = myContextFactory.getContext()

    ctx.set_verify(
        SSL.VERIFY_PEER | SSL.VERIFY_FAIL_IF_NO_PEER_CERT,
        verifyCallback
        )

    # Since we have self-signed certs we have to explicitly
    # tell the server to trust them.
    ctx.load_verify_locations("keys/ca.pem")

    reactor.listenSSL(8000, factory, myContextFactory)
    reactor.run()
```

Use the `set_verify` method to set the verification mode for a context object and the verification callback. The mode is either `VERIFY_NONE` or `VERIFY_PEER`. If `VERIFY_PEER` is set, the mode can be augmented by `VERIFY_FAIL_IF_NO_PEER_CERT` and/or `VERIFY_CLIENT_ONCE`.

The callback takes as its arguments a connection object, X509 object, error number, error depth, and return code. The purpose of the callback is to allow you to enforce additional restrictions on the verification. Thus, if the return code is False, you should return False; if the return code is True and further verification passes, return True.

Client with certificates

```
from OpenSSL import SSL
from twisted.internet import ssl, reactor
from twisted.internet.protocol import ClientFactory, Protocol

class EchoClient(Protocol):
    def connectionMade(self):
        print "hello, world"
        self.transport.write("hello, world!")

    def dataReceived(self, data):
        print "Server said:", data
        self.transport.loseConnection()

class EchoClientFactory(ClientFactory):
    protocol = EchoClient

    def clientConnectionFailed(self, connector, reason):
        print "Connection failed - goodbye!"
```

```
        reactor.stop()

    def clientConnectionLost(self, connector, reason):
        print "Connection lost - goodbye!"
        reactor.stop()

class CtxFactory(ssl.ClientContextFactory):
    def getContext(self):
        self.method = SSL.SSLv23_METHOD
        ctx = ssl.ClientContextFactory.getContext(self)
        ctx.use_certificate_file('keys/client.crt')
        ctx.use_privatekey_file('keys/client.key')

        return ctx

if __name__ == '__main__':
    factory = EchoClientFactory()
    reactor.connectSSL('localhost', 8000, factory, CtxFactory())
    reactor.run()
```

3.3.5 Other facilities

`twisted.protocols.amp` supports encrypted connections and exposes a `startTLS` method one can use or subclass. `twisted.web` has built-in SSL support in its `client`, `http`, and `xmlrpc` modules.

3.3.6 Conclusion

After reading through this tutorial, you should be able to:

- Use `listenSSL` and `connectSSL` to create servers and clients that use SSL
- Use `startTLS` to switch a channel from being unencrypted to using SSL mid-connection
- Add server and client support for client authentication

3.4 UDP Networking

3.4.1 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.IReactor` `UDP` reactor API.

3.4.2 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:

```
from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor

class Echo(DatagramProtocol):
```

```
    def datagramReceived(self, data, (host, port)):
        print "received %r from %s:%d" % (data, host, port)
        self.transport.write(data, (host, port))

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 the `host` argument should be an IP address, not a hostname. If you only have the hostname use `reactor.resolve()` to resolve the address (see `twisted.internet.interfaces.IReactorCore.resolve`).

3.4.3 Connected UDP

A connected UDP socket is slightly different from a standard one - it can only send and receive datagrams to/from a single address, but this does not in any way imply a connection. 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 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 it 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'.

```
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("hello") # no need for address

    def datagramReceived(self, data, (host, port)):
        print "received %r from %s:%d" % (data, host, port)

    # 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.:

```
from twisted.internet import reactor

def gotIP(ip):
    print "IP of 'example.com' is", ip
    reactor.callLater(3, reactor.stop)
```

```
reactor.resolve('example.com').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.

3.4.4 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.

```
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 == "Client: Ping":
            # Rather than replying to the group multicast address, we send the
            # reply directly (unicast) to the originating port:
            self.transport.write("Server: Pong", address)


# We use listenMultiple=True so that we can run MulticastServer.py and
# MulticastClient.py on same machine:
reactor.listenMulticast(8005, MulticastPingPong(),
                        listenMultiple=True)
reactor.run()
```

Source listing — *MulticastServer.py*

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.

```
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 228.0.0.5:8005 - all listeners on the multicast address
        # (including us) will receive this message.
        self.transport.write('Client: Ping', ("228.0.0.5", 8005))

    def datagramReceived(self, datagram, address):
        print "Datagram %s received from %s" % (repr(datagram), repr(address))


reactor.listenMulticast(8005, MulticastPingClient(), listenMultiple=True)
reactor.run()
```

Source listing — *MulticastClient.py*

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.

3.5 Using Processes

3.5.1 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`

3.5.2 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`.

3.5.3 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 receieved 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()
```

3.5.4 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.

3.5.5 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.

3.5.6 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.

```
#!/usr/bin/env python

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

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()
```

Source listing — *process.py*

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.
%
```

3.5.7 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:

```
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()
```

Source listing — *quotes.py*

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:

```
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()
```

Source listing — *trueandfalse.py*

3.5.8 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 addtional 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 desireable (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).

3.6 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.

3.6.1 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 asychronous 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 asychronous request (known as a series of *errbacks* or an *errback chain*). The asychronous 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.

3.6.2 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(x):
    """
    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.
    """
    d = defer.Deferred()
    # simulate a delayed result by asking the reactor to fire the
    # Deferred in 2 seconds time with the result x * 3
    reactor.callLater(2, d.callback, x * 3)
    return d

def printData(d):
    """
    Data handling function to be added as a callback: handles the
    data by printing the result
    """
    print d

d = getDummyData(3)
d.addCallback(printData)

# 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
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 x % 2 == 0:
            self.d.callback(x*3)
```

```
        else:
            self.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 printData(d):
    print d

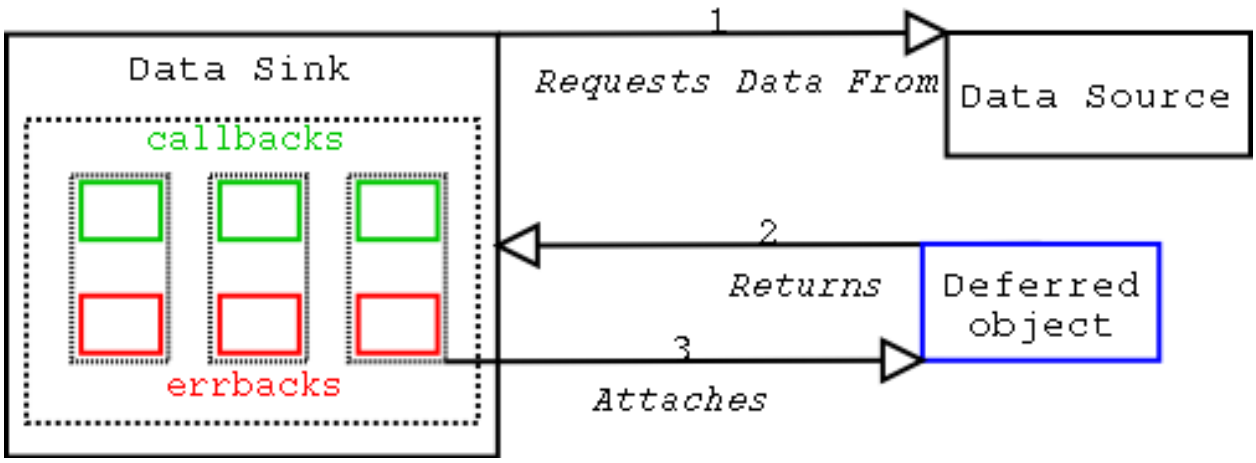
def printError(failure):
    import sys
    sys.stderr.write(str(failure))

# this series of callbacks and errbacks will print an error message
g = Getter()
d = g.getDummyData(3)
d.addCallback(printData)
d.addErrback(printError)

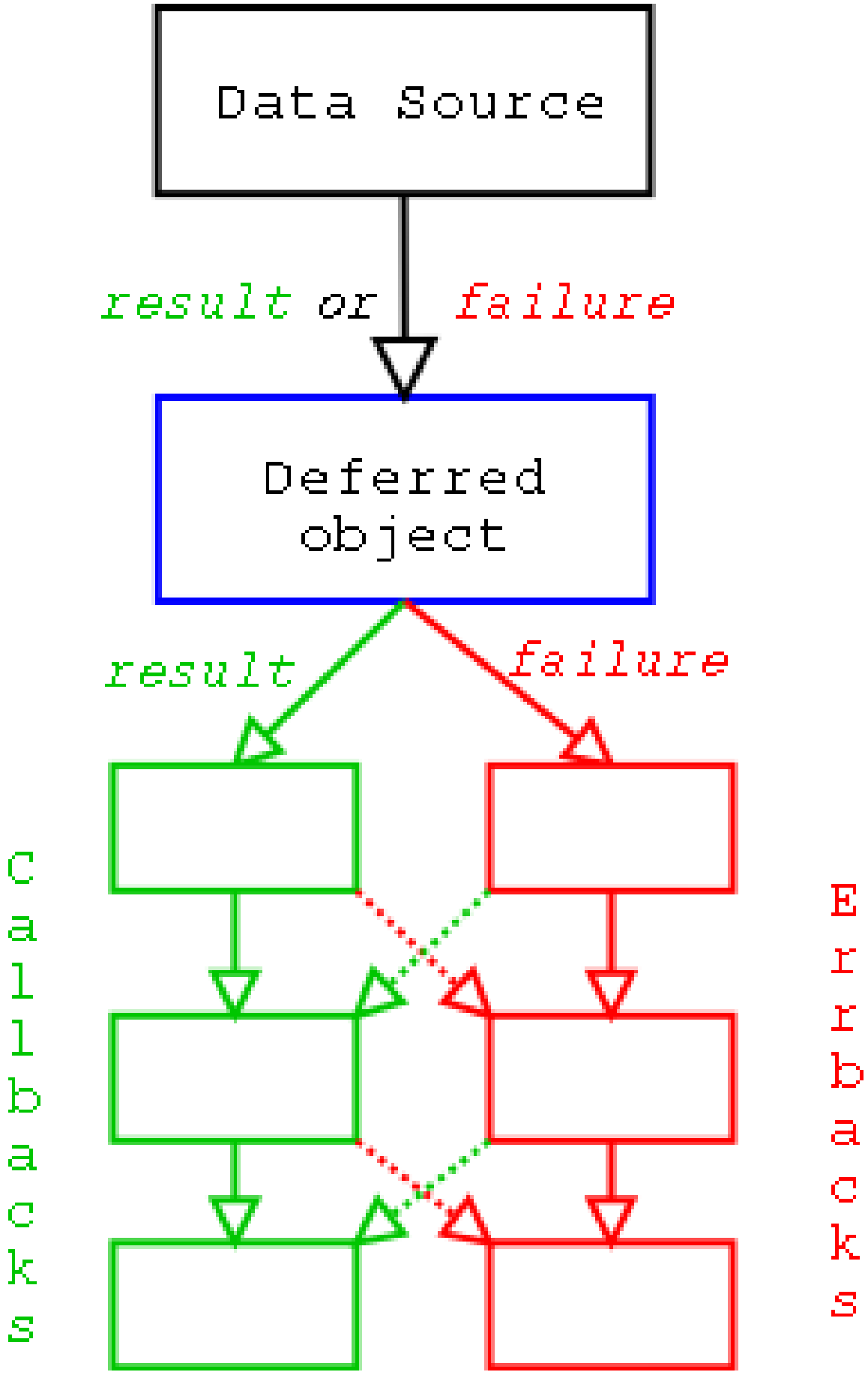
# this series of callbacks and errbacks will print "Result: 12"
g = Getter()
d = g.getDummyData(4)
d.addCallback(printData)
d.addErrback(printError)

reactor.callLater(4, reactor.stop)
reactor.run()
```

Visual Explanation



1. Requesting method (data sink) requests data, gets Deferred object.
2. Requesting method attaches callbacks to Deferred object.


Data Source
result or failure
Deferred object
result
failure
callbacks
Errbacks

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.

3.6.3 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 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 returns 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)
```

3.6.4 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 accept 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`:

```
def synchronousIsValidUser(user):
    '''
    Return true if user is a valid user, false otherwise
    '''
    return user in ["Alice", "Angus", "Agnes"]
```

Source listing — *synch-validation.py*

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* (page 145) for more information.

3.6.5 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[3]. 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 printResult(result):
    print result
d.addCallback(printResult)
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`* (page 142): 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.

3.6.6 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* (page 145).

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, *methodArgs, **methodKeywords)`. 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.

[3] 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.

- `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, **callbackOrErrback Keywords)`
  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.

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)`

3.6.7 See also

1. *Generating Deferreds* (this page), an introduction to writing asynchronous functions that return Deferreds.

3.7 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* (page 135) .

3.7.1 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* (page 144).

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.

3.7.2 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 xrange(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.

3.7.3 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.

3.7.4 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 possiblity that in the future your function might need to be asynchronous.

In the *Using Deferreds* (page 135) reference, we gave the following example of a synchronous function:

```
def synchronousIsValidUser(user):
    '''
    Return true if user is a valid user, false otherwise
    '''
    return user in ["Alice", "Angus", "Agnes"]
```

Source listing — *synch-validation.py*

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.

3.7.5 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 xrange(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()
```

3.7.6 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!

3.8 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

def runEverySecond():
    print "a second has passed"

l = task.LoopingCall(runEverySecond)
l.start(1.0) # call every second

# l.stop() will stop the looping calls
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()`.

3.9 Using Threads in Twisted

3.9.1 Running code in a thread-safe manner

Most code in Twisted is not thread-safe. For example, writing data to a transport from a protocol is not thread-safe. Therefore, we want a way to schedule methods to be run in the main event loop. This can be done using the function `twisted.internet.interfaces.IReactorThreads.callFromThread`:

```
from twisted.internet import reactor

def notThreadSafe(x):
     """do something that isn't thread-safe"""
     # ...

def threadSafeScheduler():
    """Run in thread-safe manner."""
    reactor.callFromThread(notThreadSafe, 3) # will run 'notThreadSafe(3)'
                                             # in the event loop
reactor.run()
```

3.9.2 Running code in threads

Sometimes we may want to run methods in threads - for example, in order to access blocking APIs. Twisted provides methods for doing so using the IReactorThreads API (`twisted.internet.interfaces.IReactorThreads`). Additional utility functions are provided in `twisted.internet.threads`. Basically, these methods allow us to queue methods to be run by a thread pool.

For example, to run a method in a thread we can do:

```
from twisted.internet import reactor

def aSillyBlockingMethod(x):
    import time
    time.sleep(2)
    print x

# run method in thread
reactor.callInThread(aSillyBlockingMethod, "2 seconds have passed")
reactor.run()
```

3.9.3 Utility Methods

The utility methods are not part of the `twisted.internet.reactor` APIs, but are implemented in `twisted.internet.threads`.

If we have multiple methods to run sequentially within a thread, we can do:

```
from twisted.internet import reactor, threads

def aSillyBlockingMethodOne(x):
    import time
    time.sleep(2)
    print x

def aSillyBlockingMethodTwo(x):
    print x

# run both methods sequentially in a thread
commands = [(aSillyBlockingMethodOne, ["Calling First"], {})]
commands.append((aSillyBlockingMethodTwo, ["And the second"], {}))
threads.callMultipleInThread(commands)
reactor.run()
```

For functions whose results we wish to get, we can have the result returned as a Deferred:

```
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()
```

If you wish to call a method in the reactor thread and get its result, you can use `blockingCallFromThread`:

```
from twisted.internet import threads, reactor, defer
from twisted.web.client import getPage
from twisted.web.error import Error

def inThread():
    try:
        result = threads.blockingCallFromThread(
            reactor, getPage, "http://twistedmatrix.com/")
```

```
    except Error, 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.

3.9.4 Managing the Thread Pool

The thread pool is implemented by `twisted.python.threadpool.ThreadPool`.

We may want to modify the size of the threadpool, increasing or decreasing the number of threads in use. We can do this do this quite easily:

```
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 5 and a maximum size of 10. Be careful that you understand threads and their resource usage before drastically altering the thread pool sizes.

3.10 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.

3.10.1 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.

3.10.2 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.

3.10.3 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 two interfaces `IConsumer` and `IFinishableConsumer`.

The slightly simpler of these two interfaces, `IConsumer`, defines three methods: `registerProducer`, `unregisterProducer`, and `write`. `IFinishableConsumer` adds `finish`.

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 `resume Producing` is called on them.

finish()

This method of `IFinishableConsumers` gives producers a way to explicitly notify the consumer that they have generated all the data they will ever generate.

3.10.4 Further Reading

An example push producer application can be found in `doc/examples/streaming.py`.

- *Components: Interfaces and Adapters* (page 161)
- `FileSender`: A Simple Pull Producer

3.11 Choosing a Reactor and GUI Toolkit Integration

3.11.1 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* (page 156) is Twisted's default on Linux. Other platforms use *poll()* (page 155), or the most cross-platform reactor, *select()* (page 155).

Platform-specific reactor implementations exist for:

- *Poll for Linux* (page 155)
- *Epoll for Linux 2.6* (page 156)
- *WaitForMultipleObjects (WFMO) for Win32* (page 155)
- *Input/Output Completion Port (IOCP) for Win32* (page 156)
- *KQueue for FreeBSD and Mac OS X* (page 155)
- *CoreFoundation for Mac OS X* (page 157)

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* (page 156)
- *GTK+ 3.0 and GObject Introspection* (page 156)
- *Tkinter* (page 157)
- *wxPython* (page 157)
- *Win32* (page 155)
- *CoreFoundation* (page 157)
- *PyUI* (page 158)

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.

	Status	TCP	SSL	UDP	Threading	Processes	Scheduling
select()	Stable	Y	Y	Y	Y	Y	Y
poll	Stable	Y	Y	Y	Y	Y	Y
WaitForMultipleObjects (WFMO) for Win32	Experimental	Y	Y	Y	Y	Y	Y
Input/Output Completion Port (IOCP) for Win32	Experimental	Y	Y	Y	Y	Y	Y
CoreFoundation	Unmaintained	Y	Y	Y	Y	Y	Y
epoll	Stable	Y	Y	Y	Y	Y	Y
GTK+	Stable	Y	Y	Y	Y	Y	Y
wx	Experimental	Y	Y	Y	Y	Y	Y
kqueue	Experimental	Y	Y	Y	Y	Y	Y

Table 3.1: Summary of reactor features

3.11.2 Reactor Functionality

3.11.3 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
```

3.11.4 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
```

3.11.5 GUI Integration Reactors

GTK+

Twisted integrates with PyGTK[4] 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[5] and GObject through PyGObject's[6] 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:

[4] http://www.pygtk.org/
[5] http://gtk.org
[6] http://live.gnome.org/PyGObject

```
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* (this page), the support for integrating Twisted with a wxPython[7] 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[8] 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
```

3.11.6 Non-Reactor GUI Integration

Tkinter

The support for Tkinter[9] doesn't use a specialized reactor. Instead, there is some specialized support code:

[7] http://www.wxpython.org
[8] http://pyobjc.sf.net/
[9] http://wiki.python.org/moin/TkInter

```
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* (page 157), the support for integrating Twisted with a PyUI[10] 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`.

[10] http://pyui.sourceforge.net

Chapter 4

High-Level Infrastructure

4.1 Getting Connected with Endpoints

4.1.1 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.

4.1.2 Constructing and Using Endpoints

In both *Writing Servers* (page 8) and *Writing Clients* (page 13), 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* (page 135), 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

`IStreamClientEndpoint.connect` will connect your protocol factory to a new outgoing connection attempt. It returns a `Deferred` which fires with the `IProtocol` returned from the factory's `buildProtocol` method.

Connection attempts may fail, and so that `Deferred` may also errback. If it does so, you will have to try again; your protocol won't be constructed, and no further attempts will be made.

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`, 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 = myEndpoint.connect(myFactory)
reactor.callLater(30, attempt.cancel)
```

4.1.3 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* (page 173) . 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.

4.1.4 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`.
- SSL. 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 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`.
- 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`.

Servers

- TCP. Supported arguments: port, interface, backlog. interface and backlog are optional. interface is an IP address to bind to. For example, `tcp:port=80:interface=192.168.1.1`.
- SSL. All TCP arguments are supported, plus: certKey, privateKey, and sslmethod. certKey (optional, defaults to the value of privateKey) gives a filesystem path to a certificate (PEM format). privateKey gives a filesystem path to a a private key (PEM format). sslmethod indicates which SSL/TLS version to use (a value like TLSv1_METHOD). For example, `ssl:port=443:privateKey=/etc/ssl/server.pem:sslmethod=SSLv3_METHOD`.
- 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`.

4.2 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.

4.2.1 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 implements

class AmericanSocket:

    implements(IAmericanSocket)

    def voltage(self):
        return 120
```

So, to declare that a class implements an interface, we simply call `zope.interface.implements` at the class level.

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 implements

class AdaptToAmericanSocket:

    implements(IAmericanSocket)

    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, implements
from twisted.python import components

class IAmericanSocket(Interface):
    def voltage():
      """Return the voltage produced by this socket object, as an integer.
      """


class AmericanSocket:
    implements(IAmericanSocket)

    def voltage(self):
        return 120

class UKSocket:
    def voltage(self):
        return 240

class AdaptToAmericanSocket:

    implements(IAmericanSocket)

    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 implements

class Root(Referenceable):
    implements(IPBRoot)
```

Suppose you have your own class which implements your `IMyInterface` interface:

```
from zope.interface import implements, Interface

class IMyInterface(Interface):
    pass

class MyThing:
    implements(IMyInterface)
```

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 implements, Interface

class IMyInterface(Interface):
    pass

class MyThing(pb.Root):
    implements(IMyInterface)
```

```
>>> 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 `implementOnly`:

```
from twisted.spread import pb
from zope.interface import implementsOnly, Interface

class IMyInterface(Interface):
    pass

class MyThing(pb.Root):
    implementsOnly(IMyInterface)
```

```
>>> from twisted.spread.pb import IPBRoot
>>> IPBRoot.implementedBy(MyThing)
False
```

4.3 Cred: Pluggable Authentication

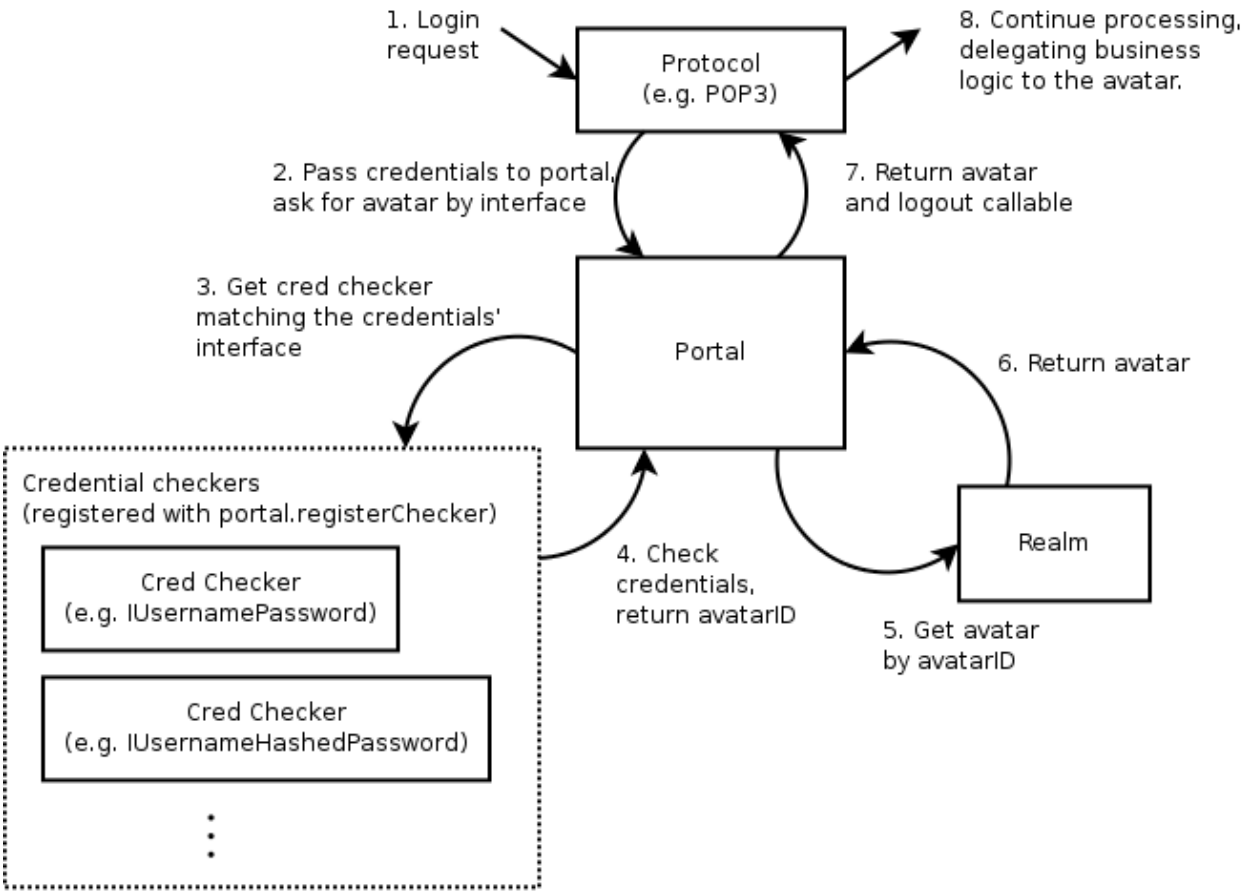
4.3.1 Goals

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:



4.3.2 Cred objects

The Portal

This is the 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.)

4.3.3 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:

```
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(
                cred.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, she 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 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"


class SimpleRealm:
    implements(IRealm)

    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()
```

4.3.4 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* (page 181) 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 implements

from twisted import plugin
from twisted.cred.strcred import ICheckerFactory
from myapp.cred import SpecialChecker

class SpecialCheckerFactory(object):
    """
    A checker factory for a specialized (fictional) API.
    """
```

```
    # The class needs to implement both of these interfaces
    # for the plugin system to find our factory.
    implements(ICheckerFactory, plugin.IPlugin)

    # 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(**dict)

# 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* (page 181) document.

4.3.5 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

4.4 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).

4.4.1 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.

4.4.2 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 implements
from twisted.plugin import IPlugin
from matsim import imatsim

class SimpleMaterial(object):
    implements(IPlugin, imatsim.IMaterial)

    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.

4.4.3 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.

4.4.4 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.

4.4.5 Further Reading

- *Components: Interfaces and Adapters* (page 161)

Chapter 5

Deploying Twisted Applications

5.1 The Basics

5.1.1 Application

Twisted programs usually work with `twisted.application.service.Application`. This class usually holds all persistent configuration of a running server – ports to bind to, places where connections to must be kept or attempted, periodic actions to do and almost everything else. It is the root object in a tree of services implementing `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`.

`Application`s are just Python objects, which can be created and manipulated in the same ways as any other object.

5.1.2 twistd

The Twisted Daemon is a program that knows how to run Applications. This program is `twistd(1)`. 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. `twistd(1)`, however, supplies many options which are highly useful for program set up.

`twistd` supports choosing a reactor (for more on reactors, see *Choosing a Reactor* (page 154)), logging to a logfile, daemonizing 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 `doc/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 (usually you would do `kill 'cat twistd.pid'`).

As always, the gory details are in the manual page.

5.1.3 OS Integration

If you have an Application that runs with `twistd`, you can easily deploy it on RedHat Linux or Debian GNU/Linux based systems using the `tap2deb` or `tap2rpm` tools. These take a Twisted Application file (of any of the supported formats Python source, XML or pickle), and build a Debian or RPM package (respectively) that installs the Application as a system service. The package includes the Application file, a default `/etc/init.d/` script that starts and stops the process with twistd, and post-installation scripts that configure the Application to be run in the appropriate init levels.

> **Note:** `tap2rpm` and `tap2deb` do not package your entire application and dependent code, just the Twisted Application file. You will need to find some other way to package your Python code, such as distutils[1]' `bdist_rpm` command.

[1] http://docs.python.org/library/distutils.html

For more savvy users, these tools also generate the source package, allowing you to modify and polish things which automated software cannot detect (such as dependencies or relationships to virtual packages).

5.2 Using the Twisted Application Framework

5.2.1 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* (page 8) and/or *clients* (page 13).

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.

5.2.2 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* (page 154) 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 `IServiceCollection`s.

Here's a simple example of constructing an Application object which represents an echo server that runs on TCP port 7001.

```
from twisted.application import internet, service
from somemodule import EchoFactory

port = 7001
factory = EchoFactory()

# this is the important bit
application = service.Application("echo")  # create the Application
echoService = internet.TCPServer(port, factory) # create the service
# add the service to the application
echoService.setServiceParent(application)
```

See *Writing Servers* (page 8) for an explanation of EchoFactory.

This example creates a simple hierarchy:

```
application
|
`- echoService
```

More complicated hierarchies of services can be created using IServiceCollection. You will most likely want to do this 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.

5.2.3 Using 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:

```
# 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)
```

Source listing — *service.tac*

`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.python.log import FileLogObserver

def logger():
    return FileLogObserver(open("/tmp/my.log", "w")).emit
```

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.python.log import ILogObserver, FileLogObserver
from twisted.python.logfile import DailyLogFile

application = Application("myapp")
logfile = DailyLogFile("my.log", "/tmp")
application.setComponent(ILogObserver, FileLogObserver(logfile).emit)
```

invoking `twistd -y my.tac` will create a log file at `/tmp/my.log`.

Services provided by Twisted

Twisted provides several services that you want to know about.

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`

`UDPClient` Services which allow you to send and receive data over UDP

- `listenUDP`

See also the *UDP documentation* (page 125).

`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.

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)
```

5.3 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* (page 173) is recommended, in particular the "Extending an Existing Program" section.
- The *Application* (page 178) 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* (page 193).

5.3.1 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.

5.3.2 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* (page 178).

5.3.3 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 implements

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."]]


class MyServiceMaker(object):
    implements(IServiceMaker, IPlugin)
    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.

5.3.4 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* (page 167) 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"]]

class MyServerServiceMaker(object):
    implements(IServiceMaker, IPlugin)
    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
```

5.3.5 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

Chapter 6

Utilities

6.1 Logging with twisted.python.log

6.1.1 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* (page 179) 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[1] 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[2] 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.

6.1.2 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.

[1] http://docs.python.org/library/logging.html
[2] http://docs.python.org/library/logging.html

6.1.3 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* (page 178) for more information.

6.2 Symbolic Constants

6.2.1 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.

6.2.2 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 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>]
>>>
```

And constants can also be compared, either 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
>>>
```

Custom functionality can also be associated with constants defined this way. A subclass of `Names` may define class methods to implement such functionality. Consider this redefinition 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
>>>
```

6.2.3 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'
>>>
```

And 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>]
>>>
```

And constants can be compared for equality and identity:

```
>>> STATUS.OK == STATUS.OK
True
>>> STATUS.OK is STATUS.OK
True
>>> STATUS.OK == STATUS.OK
True
>>> STATUS.OK is STATUS.NOT_FOUND
False
>>> STATUS.OK == STATUS.NOT_FOUND
False
>>>
```

And, as with `Names`, a subclass of `Values` can define 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 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
>>>
```

6.2.4 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
>>>
```

And 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>
>>>
```

The unary operator `~` (*not*) is also defined:

```
>>> ~Permission.USER_READ
<Permission={GROUP_EXECUTE,GROUP_READ,GROUP_WRITE,OTHER_EXECUTE,OTHER_READ,OTHER_WRITE,USE
>>>
```

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
'rw-r--r--'
>>>
```

6.3 twisted.enterprise.adbapi: Twisted RDBMS support

6.3.1 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.

6.3.2 What you should already know

- Python :-)
- How to write a simple Twisted Server (see *this tutorial* (page 8) to learn how)
- Familiarity with using database interfaces (see the documentation for DBAPI 2.0[3] or this article[4] by Andrew Kuchling)

[3] http://www.python.org/dev/peps/pep-0249/
[4] http://www.amk.ca/python/writing/DB-API.html

6.3.3 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[5]-compliant module.

`enterprise.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.

6.3.4 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 `enterprise.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 `twisted.internet.defer.Deferred`, which allows arbitrary callbacks to be called upon completion (or upon failure). More documentation on Deferred is available *here* (page 135).

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 `twisted.enterprise.adbapi.Transaction`, which basically mimics a DB-API cursor. In all cases a database transaction will be commited after your database usage is finished, unless an exception is raised in which case it will be rolled back.

[5] http://www.python.org/dev/peps/pep-0249/

```
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 paramater munging – `runQuery(query, params, ...)` maps directly onto `cursor.execute(query, params, ...)`.

6.3.5 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

# Gadfly
cp = adbapi.ConnectionPool("gadfly", "test", "/tmp/gadflyDB")

# PostgreSQL PyPgSQL
cp = adbapi.ConnectionPool("pyPgSQL.PgSQL", database="test")

# MySQL
cp = adbapi.ConnectionPool("MySQLdb", db="test")
```

6.3.6 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.

6.4 Parsing command-lines with usage.Options

6.4.1 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 her own loops, or have arbitrary limitations on the command line (the most common one being not being able to have more then one instance of a specific option, thus rendering the idiom `program -v -v -v` impossible), Twisted allows the programmer to decide how much control she wants.

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
```

6.4.2 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:

```
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, errortext:
    print '%s: %s' % (sys.argv[0], errortext)
    print '%s: Try --help for usage details.' % (sys.argv[0])
    sys.exit(1)
if options['fast']:
    print "fast",
if options['good']:
    print "good",
if options['cheap']:
    print "cheap",
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(BaseOptions):

    optFlags = [
        ["quiet", "q", "Silence output"],
        ["fast", "f", "Run quickly"],
        ["good", "g", "Don't validate input"],
        ["cheap", "c", "Use cheap resources"]
    ]
```

6.4.3 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 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, errortext:
    print '%s: %s' % (sys.argv[0], errortext)
    print '%s: Try --help for usage details.' % (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",
if config['good']:
    print "good",
if config['cheap']:
    print "cheap",
print
```

Like `optFlags`, `optParameters` works smoothly with inheritance.

6.4.4 Option Subcommands

It is useful, on occassion, 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.

6.4.5 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 rathers 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.

6.4.6 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)`.

6.4.7 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"
```

6.4.8 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.
```

6.4.9 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 Completions. See that class for further details.

In addition, `compData` may be defined on parent classes in your inheritance hiearchy. The information from each Completions instance will be aggregated when producing the final tab-completion results.

6.5 DirDBM: Directory-based Storage

6.5.1 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()
[]
```

6.5.2 dirdbm.Shelf

Sometimes it is neccessary 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 neccessary to explictly 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]
```

6.6 Writing tests for Twisted code using Trial

6.6.1 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.

To run all the Twisted tests, do:

```
$ trial twisted
```

Refer to the Trial man page for other command-line options.

6.6.2 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.

6.6.3 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 Subversion 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[6]) 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.

[6] http://docs.python.org/using/cmdline.html#cmdoption-unittest-discover-W

Chapter 7

Asynchronous Messaging Protocol (AMP)

7.1 Asynchronous Messaging Protocol 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.

7.1.1 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:

```
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)
```

Source listing — *basic_server.tac*

And you can connect to an AMP server using a client endpoint:

```
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)
    factory = Factory()
    factory.protocol = AMP
    return endpoint.connect(factory)


def main():
    startLogging(stdout)

    d = connect()
    d.addErrback(err, "Connection failed")
    def done(ignored):
        reactor.stop()
    d.addCallback(done)

    reactor.run()
```

Source listing — *basic_client.py*

7.1.2 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.

```
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()
```

Source listing — *command_client.py*

7.1.3 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.

7.1.4 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 implements

from twisted.protocols.amp import IBoxReceiver

class BoxReflector(object):
    implements(IBoxReceiver)

    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. `amp BoxReceived` 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`.

Chapter 8

Perspective Broker

8.1 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.

8.1.1 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.

8.2 Introduction to Perspective Broker

8.2.1 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[2].

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.

[2] 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.

8.2.2 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.

8.2.3 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 posess 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.

8.2.4 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.

8.3 Using Perspective Broker

8.3.1 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.

```
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()
```

Source listing — *pbsimple.py*

```
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()
```

Source listing — *pbsimpleclient.py*

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 `PBServer Factory` 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 `.getRoot Object()`.

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* (page 135) 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. `call Remote()` *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 `Remote Reference` 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[3]. 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`.)

8.3.2 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.

```
#!/usr/bin/env python

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

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()
```

Source listing — *pb1server.py*

```
#!/usr/bin/env python

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

from twisted.spread import pb
from twisted.internet import reactor

def main():
    factory = pb.PBClientFactory()
```

[3]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.

```
    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()
```

Source listing — *pb1client.py*

`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[4].

8.3.3 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()`[5].

```
#!/usr/bin/env python

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

from twisted.spread import pb
```

[4]This can be overridden, by subclassing one of the Serializable flavors and defining custom serialization code for your class. See *Passing Complex Types* (page 223) for details.

[5]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.

```
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)
reactor.listenTCP(8800, pb.PBServerFactory(root_obj))
reactor.run()
```

Source listing — *pb2server.py*

```
#!/usr/bin/env python

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

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()
```

Source listing — *pb2client.py*

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 are added with `.addCallback` instead of `.addCallbacks`. As you can tell from the *Deferred* (page 135) 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.

8.3.4 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.

```
#!/usr/bin/env python

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

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()
```

Source listing — *pb3server.py*

```
#!/usr/bin/env python

# Copyright (c) Twisted Matrix Laboratories.
```

```
# See LICENSE for details.

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()
```

Source listing — *pb3client.py*

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.

8.3.5 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[6]. 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.

```
#!/usr/bin/env python

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

from twisted.spread import pb
from twisted.internet import reactor
```

[6]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.

```
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
        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()
```

Source listing — *exc_server.py*

```
#!/usr/bin/env python

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

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()
```

Source listing — *exc_client.py*

```
$ ./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`.

8.3.6 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.

```
#!/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()
```

Source listing — *trap_server.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

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()
```

Source listing — *trap_client.py*

```
$ ./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[7].

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` 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. Kevin Turner <http://twistedmatrix.com/users/acapnotic/>

8.4 Managing Clients of Perspectives

8.4.1 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.

8.4.2 Managing Avatars

The simplest way to manage persistent avatars is to use a straight-forward caching mechanism:

```
from zope.interface import implements
```

[7]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* (page 223) for an example.

```
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)

class CachingRealm:
    implements(portal.IRealm)

    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 implements

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

class CachingRealm:
    implements(portal.IRealm)

    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 implements

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)

class CachingRealm:
    implements(portal.IRealm)

    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 implements

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)

class CachingRealm:
    implements(portal.IRealm)

    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
```

8.4.3 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 implements

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()

class Realm:
    implements(portal.IRealm)

    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.

8.5 PB Copyable: Passing Complex Types

8.5.1 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`).

8.5.2 Motivation

From the *previous chapter* (page 209), 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.

8.5.3 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 [8]. 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.

8.5.4 pb.Copyable

Ok, enough of this theory. How do you send a fully-fledged object from one side to the other?

```
#!/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

class LilyPond:
    def setStuff(self, color, numFrogs):
        self.color = color
        self.numFrogs = numFrogs
    def countFrogs(self):
        print "%d frogs" % self.numFrogs

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):
```

[8] 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 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".

```
        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()
```

Source listing — *copy_sender.py*

```
# 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.
"""

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))
```

Source listing — *copy_receiver.tac*

The sending side has a class called `LilyPond`. To make this eligble 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* (page 235)).

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.

```
#!/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)
```

Source listing — *copy2_classes.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 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()
```

Source listing — *copy2_sender.py*

```
#!/usr/bin/env python

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

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))
```

Source listing — *copy2_receiver.py*

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 "sibiling 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`[9].
- 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).

[9] `pb.RemoteCopy` is actually defined in `twisted.spread.flavors`, but `pb.RemoteCopy` is the preferred way to access it

- `twisted.manhole.explorer` also uses `Copyable` to distribute debugging information from the program under test to the debugging tool.

8.5.5 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[10].

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 with which the cache is first filled. It also provides an object called the "observer" [11] 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:

```
#!/usr/bin/env python

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

[10] 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.

[11] 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.

```
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)
    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)
```

Source listing — *cache_classes.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()
```

Source listing — *cache_sender.py*

```
#!/usr/bin/env python

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

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))
```

Source listing — *cache_receiver.py*

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[12].
- 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.

[12] 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.

- 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.
- `twisted.manhole.explorer` uses `Cacheable`, and does some fairly interesting things with it.
- The `spread.publish` module also uses `Cacheable`, and might be a source of further information.

8.6 Authentication with Perspective Broker

8.6.1 Overview

The examples shown in *Using Perspective Broker* (page 209) 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* (page 167) to handle authentication issues, and Perspective Broker has code to make it easy to implement the most common use cases.

8.6.2 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 not self.groups.has_key(groupname):
            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* (page 209). 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.[13]

(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[14], 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 not self.groups.has_key(groupname):
            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:

[13] 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.

[14] 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*.

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.

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.
>
> Futhermore, 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* (page 249).

8.6.3 Avatars and Perspectives

In Twisted's *cred* (page 167) 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[15]. Just remember that perspectives and avatars are basically the same thing.

Despite all we've been *telling you* (page 167) 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`[16]. 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* (page 167), 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.

8.6.4 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

```
#!/usr/bin/env python

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

from zope.interface import implements

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):
```

[15] 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.

[16] 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.

```
        print "I am", self.name, "perspective_foo(",arg,") called on", self

class MyRealm:
    implements(portal.IRealm)
    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()
```

Source listing — *pb5server.py*

```
#!/usr/bin/env python

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

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()
```

Source listing — *pb5client.py*

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

```
#!/usr/bin/env python
```

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

from zope.interface import implements

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

class MyRealm:
    implements(portal.IRealm)
    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()
```

Source listing — *pb6server.py*

```
#!/usr/bin/env python

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

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()
```

Source listing — *pb6client1.py*

```
#!/usr/bin/env python

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

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()
```

Source listing — *pb6client2.py*

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* (page 209)) 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

```
#!/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 sys import stdout

from zope.interface import implements

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)


class MyRealm(object):
    """
    Trivial realm which supports anonymous and named users by creating
    avatars which are instances of MyPerspective for either.
    """
    implements(IRealm)

    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()
```

Source listing — *pbAnonServer.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 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()
```

Source listing — *pbAnonClient.py*

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`.

8.6.5 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).

```
#!/usr/bin/env python

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

from zope.interface import implements

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 not self.groups.has_key(groupname):
            self.groups[groupname] = Group(groupname, allowMattress)
        self.groups[groupname].addUser(user)
        return self.groups[groupname]

class ChatRealm:
    implements(portal.IRealm)
    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()
```

Source listing — *chatserver.py*

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:

```
#!/usr/bin/env python
# Copyright (c) Twisted Matrix Laboratories.
# See LICENSE for details.

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()
```

Source listing — *chatclient.py*

8.7 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.

8.7.1 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`.

8.7.2 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.

Chapter 9

Manual Pages

9.1 TRIAL.1

9.1.1 NAME

trial - run unit tests

9.1.2 SYNOPSIS

trial [*options*] [*file — package — module — TestCase — testmethod*] ...
trial –help — -h

9.1.3 DESCRIPTION

trial loads and executes a suite of unit tests, obtained from modules, packages and files listed on the command line.

trial will take either filenames or fully qualified Python names as arguments. Thus 'trial myproject/foo.py', 'trial myproject.foo' and 'trial myproject.foo.SomeTestCase.test_method' are all valid ways to invoke trial.

After running the given test suite, the default test reporter prints a summary of the test run. This consists of the word "PASSED" (if all tests ran as expected) or "FAILED" (if any test behaved unexpectedly) followed by a count of the different kinds of test results encountered. The possible kinds of test results includes:

successes Tests that passed all their assertions and completed without error. These are marked "PASSED" in the normal test output.

failures Tests that failed an assertion, called self.fail() or explicitly raised self.failureException for some reason. These are marked "FAILED" in the normal test output.

errors Tests that raised an unexpected exception (including AssertionError), tests that caused the tearDown() method to raise an exception, tests that run for longer than the timeout interval, tests that caused something to call twisted.python.log.err() without subsequently calling self.flushLoggedErrors(), tests that leave the reactor in an unclean state, etc. These are marked "ERROR" in the normal test output. Note that because errors can be caused after the actual test method returns, it is possible for a single test to be reported as both an error and a failure, and hence the total number of test results can be greater than the total number of tests executed.

skips Tests that were skipped, usually because of missing dependencies. These are marked "SKIPPED" in the normal test output.

expectedFailures Tests that failed, but were expected to fail, usually because the test is for a feature that hasn't been implemented yet. These are marked "TODO" in the normal test output.

unexpectedSuccesses Tests that should have been listed under expectedFailures, except that for some reason the test succeeded. These are marked "SUCCESS!?!" in the normal test output.

9.1.4 OPTIONS

-b*, *–debug Run the tests in the Python debugger. Also does post-mortem debugging on exceptions. Will load '.pdbrc' from current directory if it exists.

-B*, *–debug-stacktraces Report Deferred creation and callback stack traces

–coverage Generate coverage information in the 'coverage' subdirectory of the trial temp directory ('_trial_temp' by default). For each Python module touched by the execution of the given tests, a file will be created in the coverage directory named for the module's fully-qualified name with the suffix '.cover'. For example, because the trial test runner is written in Python, the coverage directory will almost always contain a file named 'twisted.trial.runner.cover'. Each '.cover' file contains a copy of the Python source of the module in question, with a prefix at the beginning of each line containing coverage information. For lines that are not executable (blank lines, comments, etc.) the prefix is blank. For executable lines that were run in the course of the test suite, the prefix is a number indicating the number of times that line was executed. The string '>>>>>>' prefixes executable lines that were not executed in the course of the test suite. Note that this functionality uses Python's sys.settrace() function, so tests that call sys.settrace() themselves are likely to break trial's coverage functionality.

–disablegc Disable the garbage collector for the duration of the test run. As each test is run, trial saves the TestResult objects, which means that Python's garbage collector has more non-garbage objects to wade through, making each garbage-collection run slightly slower. Disabling garbage collection entirely will make some test suites complete faster (contrast –force-gc, below), at the cost of increasing (possibly greatly) memory consumption. This option also makes tests slightly more deterministic, which might help debugging in extreme circumstances.

-e*, *–rterrors Print tracebacks to standard output as soon as they occur

–force-gc Run gc.collect() before and after each test case. This can be used to isolate errors that occur when objects get collected. This option would be the default, except it makes tests run about ten times slower.

-h*, *–help Print a usage message to standard output, then exit.

–help-reporters Print a list of valid reporters to standard output, then exit. Reporters can be selected with the –reporter option described below.

–help-reactors Print a list of possible reactors to standard output, then exit. Not all listed reactors are available on every platform. Reactors can be selected with the –reactor option described below.

-l*, *–logfile logfile Direct the log to a different file. The default file is 'test.log'. *logfile* is relative to _trial_temp.

-n*, *–dry-run Go through all the tests and make them pass without running.

-N*, *–no-recurse By default, trial recurses through packages to find every module inside every subpackage. Unless, that is, you specify this option.

–nopm Don't automatically jump into debugger for post-mortem analysis of exceptions. Only usable in conjunction with –debug.

–profile Run tests under the Python profiler.

-r*, *–reactor reactor Choose which reactor to use. See –help-reactors for a list.

–recursionlimit Set Python's recursion limit. See sys.setrecursionlimit()

–reporter Select the reporter to use for trial's output. Use the –help-reporters option to see a list of valid reporters.

–spew Print an insanely verbose log of everything that happens. Useful when debugging freezes or locks in complex code.

–tbformat format Format to display tracebacks with. Acceptable values are 'default', 'brief' and 'verbose'. 'brief' produces tracebacks that play nicely with Emacs' GUD.

–temp-directory directory WARNING: Do not use this options unless you know what you are doing. By default, trial creates a directory called _trial_temp under the current working directory. When trial runs, it first *deletes* this directory, then creates it, then changes into the directory to run the tests. The log file and any coverage files are stored here. Use this option if you wish to have trial run in a directory other than _trial_temp. Be warned, trial will *delete* the directory before re-creating it.

–testmodule filename Ask trial to look into *filename* and run any tests specified using the Emacs-style buffer variable 'test-case-name'.

–unclean-warnings As of Twisted 8.0, trial will report an error if the reactor is left unclean at the end of the test. This option is provided to assist in migrating from Twisted 2.5 to Twisted 8.0 and later. Enabling this option will turn the errors into warnings.

-u, –until-failure Keep looping the tests until one of them raises an error or a failure. This is particularly useful for reproducing intermittent failures.

–version Prints the Twisted version number and exit.

–without-module modulenames Simulate the lack of the specified comma-separated list of modules. This makes it look like the modules are not present in the system, causing tests to check the behavior for that configuration.

-z, –random **[*seed*]** Run the tests in random order using the specified seed.

9.1.5 SEE ALSO

The latest version of the trial documentation can be found at http://twistedmatrix.com/documents/current/core/howto/testing.html

9.1.6 AUTHOR

Written by Jonathan M. Lange

9.1.7 REPORTING BUGS

To report a bug, visit http://twistedmatrix.com/trac/newticket

9.1.8 COPYRIGHT

9.2 TWISTD.1

9.2.1 NAME

twistd - run Twisted applications (TACs, TAPs)

9.2.2 SYNOPSIS

twistd [options]

9.2.3 DESCRIPTION

Read a twisted.application.service.Application out of a file and run it.

9.2.4 OPTIONS

-n, –nodaemon Don't daemonize (stay in foreground).

-q, –quiet No-op for backwards compatibility.

-p, –profile <***profile output***> Run the application under the profiler, dumping results to the specified file.

–profiler <***profiler name***> Specify the profiler to use. Defaults to the 'hotshot' profiler.

–savestats Save the Stats object rather than the text output of the profiler.

-b, –debug Run the application in the Python Debugger (implies *–nodaemon* option). Sending a SIGINT or SIGUSR2 signal to the process will drop it into the debugger.

-e, –encrypted <***file***> The specified tap/aos file is encrypted.

–euid Set only effective user-id rather than real user-id. This option has no effect unless the server is running as root, in which case it means not to shed all privileges after binding ports, retaining the option to regain privileges in cases such as spawning processes. Use with caution.

-o, –no_save Do not save shutdown state.

–originalname Behave as though the specified Application has no process name set, and run with the standard process name (the Python binary in most cases).

-l, –logfile <***logfile***> Log to a specified file, - for stdout (default: twistd.log). The log file will be rotated on SIGUSR1.

-l, –logger <***fully qualified python name***> A fully-qualified name to a log observer factory to use for the initial log observer. Takes precedence over –logfile and –syslog.

–pidfile <***pidfile***> Save pid in specified file (default: twistd.pid).

–chroot <***directory***> Chroot to a supplied directory before running (default: don't chroot). Chrooting is done before changing the current directory.

-d, –rundir <***directory***> Change to a supplied directory before running (default: .).

-u, –uid <***uid***> The uid to run as (default: don't change).

-g, –gid <***gid***> The gid to run as (default: don't change).

–umask <***mask***> The (octal) file creation mask to apply. (default: 0077 for daemons, no change otherwise).

-r, –reactor <***reactor***> Choose which reactor to use. See *–help-reactors* for a list of possibilities.

–help-reactors List the names of possibly available reactors.

–spew Write an extremely verbose log of everything that happens. Useful for debugging freezes or locks in complex code.

***-f*, *–file* <*tap file*>** Read the given .tap file (default: twistd.tap).

***-s*, *–source* <*tas file*>** Load an Application from the given .tas (AOT Python source) file.

***-y*, *–python* <*python file*>** Use the variable "application" from the given Python file. This option overrides *-f*. This option implies *–no_save*.

–syslog Log to syslog instead of a file.

–version Print version information and exit.

***–prefix* <*prefix*>** Use the specified prefix when logging to logfile. Default is "twisted".

Note that if *twistd* is run as root, the working directory is *not* searched for Python modules.

9.2.5 SIGNALS

A running twistd accepts SIGINT for a clean shutdown and SIGUSR1 to rotate log files.

9.2.6 AUTHOR

Written by Moshe Zadka, based on twistd's help messages.

9.2.7 REPORTING BUGS

To report a bug, visit *http://twistedmatrix.com/trac/wiki/TwistedDevelopment#DevelopmentProcess*

9.2.8 COPYRIGHT

9.3 TAP2DEB.1

9.3.1 NAME

tap2deb - create Debian packages which wrap .tap files

9.3.2 SYNOPSIS

tap2deb [options]

9.3.3 DESCRIPTION

Create a ready to upload Debian package in ".build"

-u, –unsigned do not sign the Debian package

-t, –tapfile <***tapfile***> Build the application around the given .tap (default twistd.tap)

-y, –type <***type***> The configuration has the given type . Allowable types are *tap*, *source*, *xml* and *python*. The first three types are *mktap* output formats, while the last one is a manual building of application (see *twistd(1)*, the *-y* option).

-p, –protocol <***protocol***> The name of the protocol this will be used to serve. This is intended as a part of the description. Default is the name of the tapfile, minus any extensions.

-d, –debfile <***debfile***> The name of the debian package. Default is 'twisted-'+protocol.

-V, –set-version <***version***> The version of the Debian package. The default is 1.0

-e, –description <***description***> The one-line description. Default is uninteresting.

-l, –long_description <***long_description***> A multi-line description. Default is explanation about this being an automatic package created from tap2deb.

-m, –maintainer <***maintainer***> The maintainer, as "Name Lastname <email address>". This will go in the metafiles, as well as be used as the id to sign the package.

–version Output version information and exit.

9.3.4 AUTHOR

Written by Moshe Zadka, based on twistd's help messages

9.3.5 REPORTING BUGS

To report a bug, visit *http://twistedmatrix.com/bugs/*

9.3.6 COPYRIGHT

9.4 TAP2RPM.1

9.4.1 NAME

tap2rpm - create RPM packages which wrap .tap files

9.4.2 SYNOPSIS

tap2rpm [options]

9.4.3 DESCRIPTION

Create a set of RPM/SRPM packages in the current directory

-t*, *–tapfile <***tapfile***> Build the application around the given .tap (default twistd.tap)

-y*, *–type <***type***> The configuration has the given type . Allowable types are *tap*, *source*, *xml* and *python*. The first three types are *mktap* output formats, while the last one is a manual building of application (see *twistd(1)*, the *-y* option).

-p*, *–protocol <***protocol***> The name of the protocol this will be used to serve. This is intended as a part of the description. Default is the name of the tapfile, minus any extensions.

-d*, *–rpmfile <***rpmfile***> The name of the RPM package. Default is 'twisted-'+protocol.

-V*, *–set-version <***version***> The version of the RPM package. The default is 1.0

-e*, *–description <***description***> The one-line description. Default is uninteresting.

-l*, *–long_description <***long_description***> A multi-line description. Default is explanation about this being an automatic package created from tap2rpm.

-m*, *–maintainer <***maintainer***> The maintainer, as "Name Lastname <email address>". This will go in the metafiles.

–version Output version information and exit.

9.4.4 AUTHOR

tap2rpm was written by Sean Reifschneider based on tap2deb by Moshe Zadka. This man page is heavily based on the tap2deb man page by Moshe Zadka.

9.4.5 REPORTING BUGS

To report a bug, visit *http://twistedmatrix.com/trac/wiki/TwistedDevelopment#FilingTickets* for more information.

9.4.6 COPYRIGHT

Chapter 10

Appendix

10.1 Twisted Glossary

adaptee An object that has been adapted, also called "original". See *Adapter* (this page).

`Adapter` An object whose sole purpose is to implement an Interface for another object. See *Interfaces and Adapters* (page 161).

`Application` A `twisted.application.service.Application`. There are HOWTOs on *creating and manipulating* (page 177) them as a system-administrator, as well as *using* (page 178) them in your code.

Avatar (from *Twisted Cred* (this page)) business logic for specific user. For example, in *PB* (page 260) these are perspectives, in POP3 these are mailboxes, and so on.

`Banana` The low-level data marshalling layer of *Twisted Spread* (page 261). See `twisted.spread.banana`.

`Broker` A `twisted.spread.pb.Broker`, the object request broker for *Twisted Spread* (page 261).

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* (page 161).

`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* (page 161).

`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* (page 13).

Consumer An object that consumes data from a *Producer* (page 260). See `twisted.internet.interfaces.IConsumer`.

Cred Twisted's authentication API, `twisted.cred`. See *Introduction to Twisted Cred* (page 167) and *Twisted Cred usage* (page 235).

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` A instance of `twisted.internet.defer.Deferred`, an abstraction for handling chains of callbacks and error handlers ("errbacks"). See the *Deferring Execution* (page 135) 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* (page 191) for more details.

errback A callback attached to a *Deferred* (this page) 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* (this page) instances for incoming or outgoing connections. See *Writing Servers* (page 8) and *Writing Clients* (page 13).

`Failure` Basically, an asynchronous exception that contains traceback information; these are used for passing errors through asynchronous callbacks.

im Abbreviation of "(Twisted) *Instance Messenger* (this page)".

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* (this page) with *Twisted Words* (page 261). 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* (page 161).

Jelly The serialization layer for *Twisted Spread* (page 261), although it can be used seperately 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* (page 259), in most cases). See `twisted.spread.jelly`.

Lore Lore[1] is Twisted's documentation system. The source format is a subset of XHTML, and output formats include HTML and LaTeX.

Manhole A debugging/administration interface to a Twisted application.

Microdom A partial DOM implementation using *SUX* (page 261). 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* (page 261); available from Divmod[2].

PB Abbreviation of "*Perspective Broker* (this page)".

Perspective Broker The high-level object layer of Twisted *Spread* (page 261), implementing semantics for method calling and object copying, caching, and referencing. See `twisted.spread.pb`.

Portal Glues *credential checkers* (page 259) and *realm* (this page)s together.

Producer An object that generates data a chunk at a time, usually to be processed by a *Consumer* (page 259). 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* (page 8) and *Writing Clients* (page 13).

PSU There is no PSU.

Reactor The core event-loop of a Twisted application. See *Reactor Basics* (page 119).

Reality See "*Twisted Reality* (page 261)"

realm (in *Twisted Cred* (page 259)) stores *avatars* (page 259) and perhaps general business logic. See `IRealm`.

[1] http://twistedmatrix.com/trac/wiki/TwistedLore/
[2] http://launchpad.net/nevow

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* (page 178) for a description of how they relate to *Applications* (page 259).

Spread Twisted Spread is Twisted's remote-object suite. It consists of three layers: *Perspective Broker* (page 260), *Jelly* (page 260) and *Banana*. (page 259) See *Writing Applications with Perspective Broker* (page 206).

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 *Using the Utilities* (page 177).

Trial `twisted.trial`, Twisted's unit-testing framework, based on the `unittest` standard library module. See also *Writing tests for Twisted code* (page 199).

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[3] 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* (page 193).

Words Twisted Words is a multi-protocol chat server that uses the *Perspective Broker* (page 260) 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 superceded by Divmod Nevow[4].

10.2 Debugging Python(Twisted) with Emacs

[5]

- 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 python2.3 /usr/lib/python2.3/pdb.py $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`.

[3] http://launchpad.net/imaginary
[4] http://launchpad.net/nevow
[5] POKEY THE PENGUIN IS COPYRIGHT 1998-2002 THE AUTHORS

- 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)