The Twisted Documentation

The Twisted Development Team

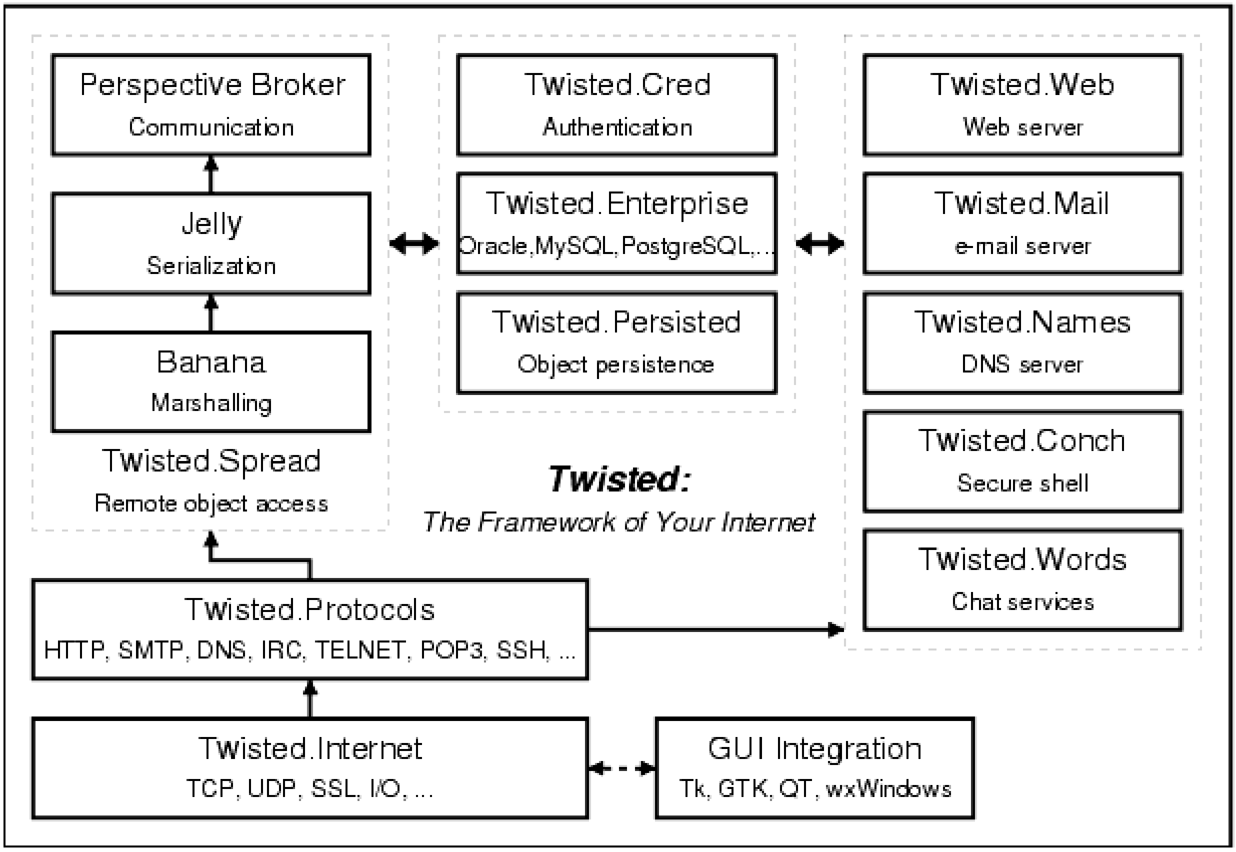
May 15, 2004

Contents

Chapter 1

Introduction

1.1 High-Level Overview of Twisted



1.2 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.

1.3 Overview of Twisted Internet

Twisted Internet is a compatible collection of 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 also contains a powerful persistence API so that network programs can be shutdown and then resurrected with most of the code unaware of this.

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` 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.

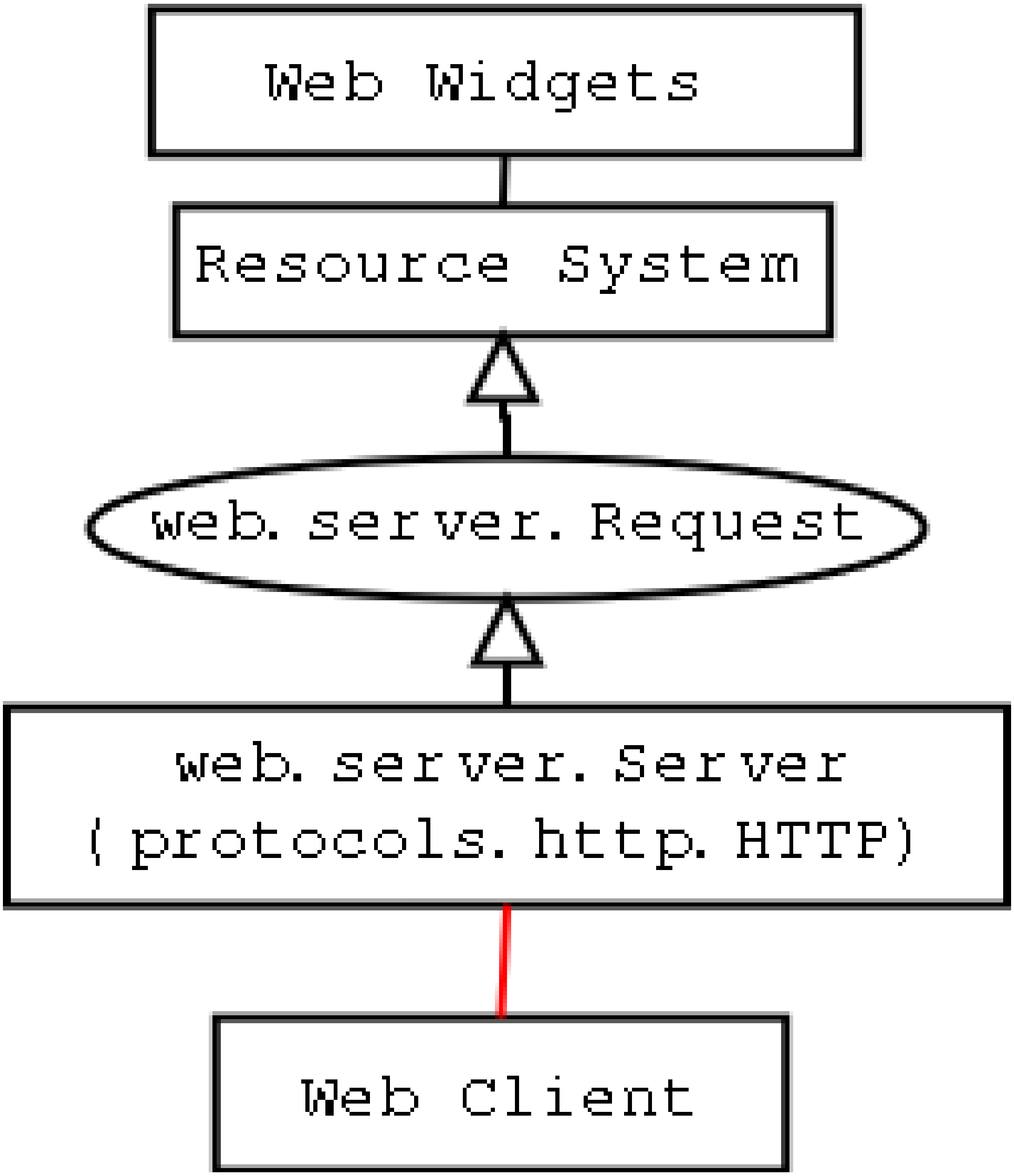
The `IProtocolFactory` dictates how factories, which are usually a large part of third party code, are written.

1.4 Overview of Twisted Web

1.4.1 Introduction

Twisted Web is a web application server written in pure Python, with APIs at multiple levels of abstraction to facilitate different kinds of web programming.

1.4.2 Twisted Web's Structure



When the Web Server receives a request from a Client, it creates a Request object and passes it on to the Resource system. The Resource system dispatches to the appropriate Resource object based on what path was requested by the client. The Resource is asked to render itself, and the result is returned to the client.

1.4.3 Resources

Resources are the lowest-level abstraction for applications in the Twisted web server. Each Resource is a 1:1 mapping with a path that is requested: you can think of a Resource as a single "page" to be rendered. The interface for making Resources is very simple; they must have a method named `render` which takes a single argument, which is the Request object (an instance of `twisted.web.server.Request`). This render method must return a string, which will be returned to the web browser making the request. Alternatively, they can return a special constant, `twisted.web.server.NOT_DONE_YET`, which tells the web server not to close the connection; you must then use `request.write(data)` to render the page, and call `request.finish()` whenever you're done.

1.4.4 Web programming with Twisted Web

Web programmers seeking a higher level abstraction than the Resource system should look at Nevow[1], available as part of the Quotient[2] project. Nevow is not distributed as part of Twisted.

Nevow is based on the same ideas as the earlier *Woven* (page 267) framework. Woven only remains part of Twisted in order to support existing projects, and should not be used for new projects. Nevow is a simpler framework which retains the strengths of Woven.

1.5 Overview of Twisted Spread

Perspective Broker (affectionately known as "PB") is an asynchronous, symmetric[3] 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.

1.5.1 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.

[1] http://www.divmod.org/Home/Projects/Nevow/

[2] http://www.divmod.org/Quotient/

[3] There is a negotiation phase for banana 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.

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.

Chapter 2

Tutorial

2.1 Twisted from Scratch, or The Evolution of Finger

2.1.1 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.

2.1.2 Contents

This tutorial is split into eleven parts:

2.2 The Evolution of Finger: building a simple finger service

2.2.1 Introduction

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

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.

2.2.2 Refuse Connections

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

Source listing — *finger01.py*

This example only runs the reactor. Nothing at all will happen until we interrupt the program. It will consume almost no CPU resources. Not very useful, perhaps – but this is the skeleton inside which the Twisted program will grow.

The Reactor

You don't call Twisted, Twisted calls you. The `reactor` is Twisted's main event loop. 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.

2.2.3 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, we start listening on port 1079. The 1079 is a reminder that eventually, we want to run on port 79, the standard port for finger servers. We define a protocol which does not respond to any events. Thus, connections to 1079 will be accepted, but the input ignored.

2.2.4 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 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.

2.2.5 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.

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

2.2.6 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, of course.

2.2.7 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.

2.2.8 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.

2.2.9 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):
        self.factory.getUser(user
        ).addErrback(lambda _: "Internal error in server"
        ).addCallback(lambda m:
                      (self.transport.write(m+"\r\n"),
                       self.transport.loseConnection()))
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 result from a remote Oracle? Or from the web? Or, or...

2.2.10 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):
        self.factory.getUser(user
        ).addErrback(lambda _: "Internal error in server"
        ).addCallback(lambda m:
                      (self.transport.write(m+"\r\n"),
                       self.transport.loseConnection()))
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*

...from running a local command? Yes, this version runs finger locally with whatever arguments it is given, and returns the standard output. 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.

2.2.11 Read Status from the Web

The web. That invention which has infiltrated homes around the world finally gets through to our invention. Here we use the built-in Twisted web client, which also returns a deferred. Finally, we manage to have examples of three different database back-ends, which do not 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):
        self.factory.getUser(user
        ).addErrback(lambda _: "Internal error in server"
        ).addCallback(lambda m:
                      (self.transport.write(m+"\r\n"),
                       self.transport.loseConnection()))
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*

2.2.12 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.

After saving the next example (finger11.py) as "finger.tac", read on to find out how to run this code using the twistd utility.

```
# 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):
        self.factory.getUser(user
        ).addErrback(lambda _: "Internal error in server"
        ).addCallback(lambda m:
                      (self.transport.write(m+"\r\n"),
                       self.transport.loseConnection()))
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.py*

2.2.13 twistd

This is how to run "Twisted Applications" – files which define an 'application'. twistd (TWISTed Daemonizer) 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 finger.tac # just like before
root% twistd -y finger.tac # daemonize, keep pid in twistd.pid
root% twistd -y finger.tac --pidfile=finger.pid
root% twistd -y finger.tac --rundir=/
root% twistd -y finger.tac --chroot=/var
root% twistd -y finger.tac -l /var/log/finger.log
root% twistd -y finger.tac --syslog # just log to syslog
root% twistd -y finger.tac --syslog --prefix=twistedfinger # use given prefix
```

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

2.3.1 Introduction

This is the second part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 17).

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.

2.3.2 Setting Message By Local Users

Now that port 1079 is free, maybe we can run on it 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):
        self.factory.getUser(user
        ).addErrback(lambda _: "Internal error in server"
        ).addCallback(lambda m:
                      (self.transport.write(m+"\r\n"),
                       self.transport.loseConnection()))

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):
        self.factory.setUser(*self.lines[:2])
        # first line: user    second line: status

class FingerSetterFactory(protocol.ServerFactory):
    protocol = FingerSetterProtocol
    def __init__(self, ff): self.setUser = ff.users.__setitem__
```

```
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.py*

2.3.3 Use Services to Make Dependencies Sane

The previous version had the setter poke at the innards of the finger factory. It's 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 moved all responsibility for manufacturing factories into the service. Note that 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.

```
# 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):
        self.factory.getUser(user
        ).addErrback(lambda _: "Internal error in server"
        ).addCallback(lambda m:
                      (self.transport.write(m+"\r\n"),
                       self.transport.loseConnection()))

class FingerSetterProtocol(basic.LineReceiver):
    def connectionMade(self): self.lines = []
    def lineReceived(self, line): self.lines.append(line)
    def connectionLost(self,reason): self.factory.setUser(*self.lines[:2])
    # first line: user   second line: status

class FingerService(service.Service):
    def __init__(self, *args, **kwargs):
        self.parent.__init__(self, *args)
        self.users = kwargs
    def getUser(self, user):
        return defer.succeed(self.users.get(user, "No such user"))
    def getFingerFactory(self):
```

```
        f = protocol.ServerFactory()
        f.protocol, f.getUser = FingerProtocol, self.getUser
        return f
    def getFingerSetterFactory(self):
        f = protocol.ServerFactory()
        f.protocol, f.setUser = FingerSetterProtocol, self.users.__setitem__
        return f

application = service.Application('finger', uid=1, gid=1)
f = FingerService('finger', 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.py*

2.3.4 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):
        self.factory.getUser(user
        ).addErrback(lambda _: "Internal error in server"
        ).addCallback(lambda m:
                      (self.transport.write(m+"\r\n"),
                       self.transport.loseConnection()))

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, f.getUser = FingerProtocol, 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.py*

2.3.5 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 (the 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.

```
# 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):
```

```
        self.factory.getUser(user
        ).addErrback(lambda _: "Internal error in server"
        ).addCallback(lambda m:
                      (self.transport.write(m+"\r\n"),
                       self.transport.loseConnection()))

class MotdResource(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):
        motd = self.users.get(username)
        username = cgi.escape(username)
        if motd is not None:
            motd = cgi.escape(motd)
            text = '<h1>%s</h1><p>%s</p>' % (username,motd)
        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._read()
    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 getFingerFactory(self):
        f = protocol.ServerFactory()
        f.protocol, f.getUser = FingerProtocol, self.getUser
        f.startService = self.startService
        return f

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

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

Source listing — *finger15.py*

2.3.6 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.protocols import basic, irc
from twisted.web import resource, server, static
import cgi
class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.factory.getUser(user
        ).addErrback(lambda _: "Internal error in server"
        ).addCallback(lambda m:
                      (self.transport.write(m+"\r\n"),
                       self.transport.loseConnection()))
class FingerSetterProtocol(basic.LineReceiver):
    def connectionMade(self): self.lines = []
    def lineReceived(self, line): self.lines.append(line)
    def connectionLost(self,reason): self.factory.setUser(*self.lines[:2])
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():
            self.factory.getUser(msg
            ).addErrback(lambda _: "Internal error in server"
            ).addCallback(lambda m: irc.IRCClient.msg(self, user, msg+': '+m))
```

```
class FingerService(service.Service):
    def __init__(self, filename):
        self.filename = filename
        self._read()
    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 getFingerFactory(self):
        f = protocol.ServerFactory()
        f.protocol, f.getUser = FingerProtocol, 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,f.nickname,f.getUser = IRCReplyBot,nickname,self.getUser
        return f

application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
serviceCollection = service.IServiceCollection(application)
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.py*

2.3.7 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.protocols import basic, irc
from twisted.web import resource, server, static, xmlrpc
import cgi
class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.factory.getUser(user
        ).addErrback(lambda _: "Internal error in server"
        ).addCallback(lambda m:
                      (self.transport.write(m+"\r\n"),
                       self.transport.loseConnection()))
class FingerSetterProtocol(basic.LineReceiver):
    def connectionMade(self): self.lines = []
    def lineReceived(self, line): self.lines.append(line)
    def connectionLost(self,reason): self.factory.setUser(*self.lines[:2])
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():
            self.factory.getUser(msg
            ).addErrback(lambda _: "Internal error in server"
            ).addCallback(lambda m: irc.IRCClient.msg(self, user, msg+': '+m))

class FingerService(service.Service):
    def __init__(self, filename):
        self.filename = filename
        self._read()
    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 getFingerFactory(self):
        f = protocol.ServerFactory()
        f.protocol, f.getUser = FingerProtocol, 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,f.nickname,f.getUser = IRCReplyBot,nickname,self.getUser
        return f

application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
serviceCollection = service.IServiceCollection(application)
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.py*

A simple client to test the XMLRPC finger:

```
# testing xmlrpc finger

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

Source listing — *fingerXRclient.py*

2.4 The Evolution of Finger: cleaning up the finger code

2.4.1 Introduction

This is the third part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 17).

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

2.4.2 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.protocols import basic, irc
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+'\n')
            self.transport.loseConnection()
        d.addCallback(writeValue)


class FingerSetterProtocol(basic.LineReceiver):

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

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

    def connectionLost(self, reason):
        self.factory.setUser(*self.lines[:2])
```

```
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._read()

    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 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

application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
serviceCollection = service.IServiceCollection(application)
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.py*

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

2.5.1 Introduction

This is the fourth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 17).

In this section of the tutorial, we'll move our code to a component architecture so that adding new features is trivial.

2.5.2 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 the __implements__ keyword. 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.protocols import basic, irc
from twisted.python import components
from twisted.web import resource, server, static, xmlrpc
import cgi

class IFingerService(components.Interface):

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

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

class IFingerSetterService(components.Interface):

    def setUser(self, 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(components.Interface):

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

    def buildProtocol(self, 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(components.Interface):

    def setUser(self, user, status):
        """Return a deferred returning a string"""

    def buildProtocol(self, 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(components.Interface):

    """
    @ivar nickname
    """

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

    def buildProtocol(self, 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._read()

    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())


application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
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 — *finger19.py*

2.5.3 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(components.Interface):

    def setUser(self, 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.protocols import basic, irc
from twisted.python import components
from twisted.web import resource, server, static, xmlrpc
import cgi

class IFingerService(components.Interface):

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

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

class IFingerSetterService(components.Interface):

    def setUser(self, 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(components.Interface):

    def getUser(self, user):
```

```
        """Return a deferred returning a string"""

    def buildProtocol(self, 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(components.Interface):

    def setUser(self, user, status):
        """Return a deferred returning a string"""

    def buildProtocol(self, 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(components.Interface):

    """
    @ivar nickname
    """

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

    def buildProtocol(self, 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.py*

2.5.4 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!

2.6 The Evolution of Finger: pluggable backends

2.6.1 Introduction

This is the fifth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 17).

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 36). 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.

2.6.2 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.protocols import basic, irc
from twisted.python import components
from twisted.web import resource, server, static, xmlrpc
import cgi
import pwd

class IFingerService(components.Interface):

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

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

class IFingerSetterService(components.Interface):

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

class IFingerSetterService(components.Interface):

    def setUser(self, 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(components.Interface):

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

    def buildProtocol(self, 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(components.Interface):

    def setUser(self, user, status):
        """Return a deferred returning a string"""

    def buildProtocol(self, 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(components.Interface):

    """
    @ivar nickname
    """

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

    def buildProtocol(self, 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._read()

    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())

# 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.py*

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

2.6.3 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.protocols import basic, irc
from twisted.python import components
from twisted.web import resource, server, static, xmlrpc
import cgi
import pwd
import os

class IFingerService(components.Interface):

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

    def getUsers(self):
```

```
        """Return a deferred returning a list of strings"""

class IFingerSetterService(components.Interface):

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

class IFingerSetterService(components.Interface):

    def setUser(self, 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(components.Interface):

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

    def buildProtocol(self, 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(components.Interface):

    def setUser(self, user, status):
        """Return a deferred returning a string"""

    def buildProtocol(self, 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(components.Interface):

    """
    @ivar nickname
    """

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

    def buildProtocol(self, 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._read()

    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())

# 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.py*

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

2.7 The Evolution of Finger: a clean web frontend

2.7.1 Introduction

This is the sixth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 17).

In this part, we demonstrate adding a web frontend using the Woven templating system.

2.7.2 Use Woven

Here we convert to using Woven, instead of manually constructing HTML snippets. Woven is a sophisticated web templating system. Its main features are to disallow any code inside the HTML, and transparent integration with deferred results.

```
# Do everything properly, and componentize
from twisted.application import internet, service
from twisted.internet import protocol, reactor, defer
from twisted.protocols import basic, irc
from twisted.python import components
from twisted.web import resource, server, static, xmlrpc, microdom
from twisted.web.woven import page, model, interfaces
```

```
import cgi

class IFingerService(components.Interface):

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

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

class IFingerSetterService(components.Interface):

    def setUser(self, 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(components.Interface):

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

    def buildProtocol(self, 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(components.Interface):

    def setUser(self, user, status):
        """Return a deferred returning a string"""

    def buildProtocol(self, 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(components.Interface):

    """
    @ivar nickname
    """

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

    def buildProtocol(self, 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 UsersModel(model.MethodModel):

    def initialize(self, *args, **kwargs):
        self.service=args[0]

    def wmfactory_users(self, request):
        return self.service.getUsers()

components.registerAdapter(UsersModel, IFingerService, interfaces.IModel)

class UserStatusTree(page.Page):

    template = """<html><head><title>Users</title></head><body>
    <h1>Users</h1>
    <ul model="users" view="List">
    <li pattern="listItem"><a view="Anchor" /></li>
    </ul></body></html>"""

    def initialize(self, *args, **kwargs):
        self.service=args[0]

    def getDynamicChild(self, path, request):
        return UserStatus(user=path, service=self.service)

    def wchild_RPC2 (self, request):
        return UserStatusXR(self.service)

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


class UserStatus(page.Page):

    template='''<html><head><title view="Text" model="user"/></head>
    <body><h1 view="Text" model="user"/>
    <p model="status" view="Text" />
    </body></html>'''

    def initialize(self, **kwargs):
        self.user = kwargs['user']
        self.service = kwargs['service']

    def wmfactory_user(self, request):
        return self.user

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

```
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._read()

    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())


application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
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 — *finger20.py*

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

2.8.1 Introduction

This is the seventh part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 17).

In this part, we add a Perspective Broker service to the finger application so that Twisted clients can access the finger server.

2.8.2 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.protocols import basic, irc
from twisted.python import components
from twisted.web import resource, server, static, xmlrpc, microdom
from twisted.web.woven import page, model, interfaces
from twisted.spread import pb
import cgi

class IFingerService(components.Interface):

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

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

class IFingerSetterService(components.Interface):

    def setUser(self, 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(components.Interface):

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

    def buildProtocol(self, 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(components.Interface):

    def setUser(self, user, status):
        """Return a deferred returning a string"""

    def buildProtocol(self, 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(components.Interface):

    """
    @ivar nickname
    """

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

    def buildProtocol(self, 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 UsersModel(model.MethodModel):

    def initialize(self, *args, **kwargs):
        self.service=args[0]

    def wmfactory_users(self, request):
        return self.service.getUsers()

components.registerAdapter(UsersModel, IFingerService, interfaces.IModel)

class UserStatusTree(page.Page):

    template = """<html><head><title>Users</title></head><body>
    <h1>Users</h1>
```

```
    <ul model="users" view="List">
    <li pattern="listItem"><a view="Anchor" /></li>
    </ul></body></html>"""

    def initialize(self, *args, **kwargs):
        self.service=args[0]

    def getDynamicChild(self, path, request):
        return UserStatus(user=path, service=self.service)

    def wchild_RPC2 (self, request):
        return UserStatusXR(self.service)

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


class UserStatus(page.Page):

    template='''<html><head><title view="Text" model="user"/></head>
    <body><h1 view="Text" model="user"/>
    <p model="status" view="Text" />
    </body></html>'''

    def initialize(self, **kwargs):
        self.user = kwargs['user']
        self.service = kwargs['service']

    def wmfactory_user(self, request):
        return self.user

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


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(components.Interface):

    def remote_getUser(self, username):
        """return a user's status"""

    def remote_getUsers(self):
        """return a user's status"""

class PerspectiveFingerFromService(pb.Root):

    __implements__ = 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._read()

    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())


application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
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(8889, pb.PBServerFactory(IPerspectiveFinger(f))
                   ).setServiceParent(serviceCollection)
```

Source listing — *finger21.py*

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*

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

2.9.1 Introduction

This is the eighth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 17).

In this part, we add HTTPS support to our web frontend, showing how to have a single factory listen on multiple ports.

2.9.2 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.protocols import basic, irc
from twisted.python import components
from twisted.web import resource, server, static, xmlrpc, microdom
from twisted.web.woven import page, model, interfaces
from twisted.spread import pb
from OpenSSL import SSL
import cgi

class IFingerService(components.Interface):

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

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

class IFingerSetterService(components.Interface):

    def setUser(self, 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(components.Interface):

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

    def buildProtocol(self, 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(components.Interface):

    def setUser(self, user, status):
        """Return a deferred returning a string"""

    def buildProtocol(self, 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(components.Interface):

    """
    @ivar nickname
    """

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

    def buildProtocol(self, 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 UsersModel(model.MethodModel):

    def initialize(self, *args, **kwargs):
        self.service=args[0]

    def wmfactory_users(self, request):
        return self.service.getUsers()

components.registerAdapter(UsersModel, IFingerService, interfaces.IModel)

class UserStatusTree(page.Page):

    template = """<html><head><title>Users</title></head><body>
    <h1>Users</h1>
    <ul model="users" view="List">
    <li pattern="listItem"><a view="Anchor" /></li>
```

```
    </ul></body></html>"""

    def initialize(self, *args, **kwargs):
        self.service=args[0]

    def getDynamicChild(self, path, request):
        return UserStatus(user=path, service=self.service)

    def wchild_RPC2 (self, request):
        return UserStatusXR(self.service)

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


class UserStatus(page.Page):

    template='''<html><head><title view="Text" model="user"/></head>
    <body><h1 view="Text" model="user"/>
    <p model="status" view="Text" />
    </body></html>'''

    def initialize(self, **kwargs):
        self.user = kwargs['user']
        self.service = kwargs['service']

    def wmfactory_user(self, request):
        return self.user

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


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(components.Interface):
```

```
    def remote_getUser(self, username):
        """return a user’s status"""

    def remote_getUsers(self):
        """return a user’s status"""

class PerspectiveFingerFromService(pb.Root):

    __implements__ = 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._read()

    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())


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)
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*

2.10 The Evolution of Finger: a Twisted finger client

2.10.1 Introduction

This is the ninth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 17).

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

2.10.2 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


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

class IFingerService(components.Interface):

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

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


class IFingerFactory(components.Interface):

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

    def buildProtocol(self, 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):
        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.py*

2.11 The Evolution of Finger: making a finger library

2.11.1 Introduction

This is the tenth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 17).

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.

2.11.2 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, 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.

2.11.3 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.

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

2.12.1 Introduction

This is the eleventh part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 17).

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.

2.12.2 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 twisted.application import internet, service, strports
from twisted.internet import protocol, reactor, defer
from twisted.protocols import basic, irc
from twisted.python import components
from twisted.web import resource, server, static, xmlrpc, microdom
from twisted.web.woven import page, model, interfaces
from twisted.spread import pb
from OpenSSL import SSL
import cgi

class IFingerService(components.Interface):

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

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

class IFingerSetterService(components.Interface):

    def setUser(self, 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(components.Interface):

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

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


class FingerFactoryFromService(protocol.ServerFactory):
    __implements__ = 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(components.Interface):

    def setUser(self, user, status):
        """Return a deferred returning a string"""

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

```
class FingerSetterFactoryFromService(protocol.ServerFactory):

    __implements__ = 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(components.Interface):

    """
    @ivar nickname
    """

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

    def buildProtocol(self, addr):
        """Return a protocol"""
```

```
class IRCClientFactoryFromService(protocol.ClientFactory):

    __implements__ = 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 UsersModel(model.MethodModel):

    def initialize(self, *args, **kwargs):
        self.service=args[0]

    def wmfactory_users(self, request):
        return self.service.getUsers()

components.registerAdapter(UsersModel, IFingerService, interfaces.IModel)

class UserStatusTree(page.Page):

    template = """<html><head><title>Users</title></head><body>
    <h1>Users</h1>
    <ul model="users" view="List">
    <li pattern="listItem"><a view="Anchor" /></li>
    </ul></body></html>"""

    def initialize(self, *args, **kwargs):
        self.service=args[0]

    def getDynamicChild(self, path, request):
        return UserStatus(user=path, service=self.service)

    def wchild_RPC2 (self, request):
        return UserStatusXR(self.service)

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

```
class UserStatus(page.Page):

    template='''<html><head><title view="Text" model="user"/></head>
    <body><h1 view="Text" model="user"/>
    <p model="status" view="Text" />
    </body></html>'''

    def initialize(self, **kwargs):
        self.user = kwargs['user']
        self.service = kwargs['service']

    def wmfactory_user(self, request):
        return self.user

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


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(components.Interface):

    def remote_getUser(self, username):
        """return a user's status"""

    def remote_getUsers(self):
        """return a user's status"""

class PerspectiveFingerFromService(pb.Root):

    __implements__ = 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__ = service.Service.__implements__, IFingerService

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

    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())


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(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
```

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:

```
#finger/plugins.tml
register('finger', 'finger.tap',
description='Build a finger server tap',
type='tap', tapname='finger')
```

finger/plugins.tml — *plugins.tml*

And now, the following works

```
% mktap finger --file=/etc/users --ircnick=fingerbot
% sudo twistd -nf finger.tap
```

2.12.3 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 #[maybe other options needed]
% sudo rpm -i .build/*.rpm
```

Will properly register the tap/tac, init.d scripts, etc. for the given file.
If it doesn't work on your favorite OS: patches accepted!

Chapter 3

Low-Level Twisted

3.1 Asynchronous Programming

3.1.1 Introduction

There are many ways to write network programs. The main ones are:

1. Handle each connection in a separate process
2. Handle each connection in a separate thread[1]
3. Use non-blocking system calls to handle all connections in one thread.

When dealing with many connections in one thread, the scheduling is the responsibility of the application, not the operating system, and is usually implemented by calling a registered function when each connection is ready to for reading or writing – commonly known as asynchronous, event-driven or callback-based programming.

Multi-threaded programming is tricky, even with high level abstractions, and Python's Global Interpreter Lock[2] limits the potential performance gain. Forking Python processes also has many disadvantages, such as Python's reference counting not playing well with copy-on-write and problems with shared state. Consequently, it was felt the best option was an event-driven framework. A benefit of such an approach is that by letting other event-driven frameworks take over the main loop, server and client code are essentially the same – making peer-to-peer a reality.

However, event-driven programming still contains some tricky aspects. As each callback must be finished as soon as possible, it is not possible to keep persistent state in function-local variables. In addition, some programming techniques, such as recursion, are impossible to use – for example, this rules out protocol handlers being recursive-descent parsers. Event-driven programming has a reputation of being hard to use due to the frequent need to write state machines. Twisted was built with the assumption that with the right library, event-driven programming is easier than multi-threaded programming.

Note that Twisted still allows the use of threads if you really need them, usually to interface with synchronous legacy code. See *Using Threads* (page 134) for details.

[1] There are variations on this method, such as a limited-size pool of threads servicing all connections, which are essentially just optimizations of the same idea.

[2] http://www.python.org/doc/current/api/threads.html

3.1.2 Async Design Issues

In Python, code is often divided into a generic class calling overridable methods which subclasses implement. In that, and similar, cases, it is important to think about likely implementations. If it is conceivable that an implementation might perform an action which takes a long time (either because of network or CPU issues), then one should design that method to be asynchronous. In general, this means to transform the method to be callback based. In Twisted, it usually means returning a *Deferred* (page 120).

Since non-volatile state cannot be kept in local variables, because each method must return quickly, it is usually kept in instance variables. In cases where recursion would have been tempting, it is usually necessary to keep stacks manually, using Python's list and the `.append` and `.pop` method. Because those state machines frequently get non-trivial, it is better to layer them such that each one state machine does one thing – converting events from one level of abstraction to the next higher level of abstraction. This allows the code to be clearer, as well as easier to debug.

3.1.3 Using Reflection

One consequence of using the callback style of programming is the need to name small chunks of code. While this may seem like a trivial issue, used correctly it can prove to be an advantage. If strictly consistent naming is used, then much of the common code in parsers of the form of if/else rules or long cases can be avoided. For example, the SMTP client code has an instance variable which signifies what it is trying to do. When receiving a response from the server, it just calls the method `"do_%s_%s" % (self.state, responseCode)`. This eliminates the requirement for registering the callback or adding to large if/else chains. In addition, subclasses can easily override or change the actions when receiving some responses, with no additional harness code. The SMTP client implementation can be found in `twisted/protocols/smtp.py`.

3.2 Reactor Overview

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

3.2.1 Reactor Basics

The reactor is the core of the event loop within Twisted – the loop which drives applications using Twisted. 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 133) and *Using Deferreds* (page 120);
- the communication howtos: *TCP servers* (page 98), *TCP clients* (page 103), *UDP networking* (page 109) and *Using processes* (page 111); and
- *Using threads* (page 134).

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 136).

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 154) 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 111) document for more information.
- `IReactorSSL`: SSL networking support.
- `IReactorTCP`: TCP networking support. More information can be found in the *Writing Servers* (this page) and *Writing Clients* (page 103) documents.
- `IReactorThreads`: Threading use and management. More information can be found within *Threading In Twisted* (page 134).
- `IReactorTime`: Scheduling interface. More information can be found within *Scheduling Tasks* (page 133).
- `IReactorUDP`: UDP networking support. More information can be found within *UDP Networking* (page 109).
- `IReactorUNIX`: UNIX socket support.

3.3 Writing Servers

3.3.1 Overview

Twisted is a framework designed to be very flexible and let you write powerful servers. The cost of this flexibility is a few layers in the way to writing your server.

This document describes the `Protocol` layer, where you implement protocol parsing and handling. If you are implementing an application then you should read this document second, after first reading the top level overview of how to begin writing your Twisted application, in *Writing Plug-Ins for Twisted* (page 157). This document is only relevant to TCP, SSL and Unix socket servers, there is a *separate document* (page 109) for 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 might be instantiated per-connection, on demand, and might 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 default factory class just instantiates each `Protocol`, and then sets on it an attribute called `factory` which points to itself. This lets every `Protocol` access, and possibly modify, the persistent configuration.

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 `twisted.internet.interfaces.IReactorTCP.listenTCP`, and the other `IReactor*.listen*` APIs for more information.

This document will explain each step of the way.

3.3.2 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. What this means is 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

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 connectionMade(self):
        self.factory.numProtocols = self.factory.numProtocols+1
        if self.factory.numProtocols > 100:
            self.transport.write("Too many connections, try later")
            self.transport.loseConnection()

    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 the factory. `connectionMade` immediately closes the connection if there are too many active protocols.

Using the Protocol

In this section, I will explain how to test your protocol easily. (In order to see how you should write a production-grade Twisted server, though, you should read the *Writing Plug-Ins for Twisted* (page 157) HOWTO as well).

Here is code that will run the QOTD server discussed earlier

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

class QOTD(Protocol):

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

# Next lines are magic:
factory = Factory()
factory.protocol = QOTD

# 8007 is the port you want to run under. Choose something >1024
reactor.listenTCP(8007, factory)
reactor.run()
```

Don't worry about the last 6 magic lines – you will understand what they do later in the document.

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 again.

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.
- Use Python's dynamicity to create open ended state machines. See, for example, the code for the SMTP client.
- 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.

3.3.3 Factories

As mentioned before, usually the class `twisted.internet.protocol.Factory` works, and there is no need to subclass it. However, sometimes there can be factory-specific configuration of the protocols, or other considerations. In those cases, there is a need to subclass `Factory`.

For a factory which simply instantiates instances of a specific protocol class, simply instantiate `Factory`, and sets its `protocol` attribute:

```
from twisted.internet.protocol import Factory
from twisted.protocols.wire import Echo

myFactory = Factory()
myFactory.protocol = Echo
```

If there is a need to easily construct factories for a specific configuration, a factory function is often useful:

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

class QOTD(Protocol):

    def connectionMade(self):
        self.transport.write(self.factory.quote+'\r\n')
        self.transport.loseConnection()


def makeQOTDFactory(quote=None):
```

```
    factory = Factory()
    factory.protocol = QOTD
    factory.quote = quote or 'An apple a day keeps the doctor away'
    return factory
```

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

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

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


class LoggingProtocol(LineReceiver):

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


class LogfileFactory(Factory):

    protocol = LoggingProtocol

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

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

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

Putting it All Together

So, you know what factories are, and want to run the QOTD with configurable quote server, do you? No problems, here is an example.

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

class QOTD(Protocol):

    def connectionMade(self):
        self.transport.write(self.factory.quote+'\r\n')
        self.transport.loseConnection()
```

```
class QOTDFactory(Factory):

    protocol = QOTD

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

reactor.listenTCP(8007, QOTDFactory("configurable quote"))
reactor.run()
```

The only lines you might not understand are the last two.

`listenTCP` is the method which connects a `Factory` to the network. It uses the reactor interface, which lets many different loops handle the networking code, without modifying end-user code, like this. As mentioned above, if you want to write your code to be a production-grade Twisted server, and not a mere 20-line hack, you will want to use *the Application object* (page 154).

3.4 Writing Clients

3.4.1 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 109).

At the base, the place where you actually implement the protocol parsing and handling, is the Protocol class. This class will usually be decended 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.ClientFactory`. The default factory class just instantiate the Protocol, and then sets on it an attribute called `factory` which points to itself. This let the Protocol access, and possibly modify, the persistent configuration.

3.4.2 Protocol

As mentioned above, this, and auxiliary classes and functions, is where most of the code is. A Twisted protocol handles data in an asynchronous manner. What this means is 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 simply writes to standard output whatever it reads from the connection. 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.

3.4.3 ClientFactory

We use reactor.connect* and 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 on the event of a connection error. Here is an example of a simple ClientFactory that uses the Echo protocol (above) and also prints what state the connection is in.

```
from twisted.internet.protocol import Protocol, ClientFactory
from sys import stdout
class Echo(Protocol):

    def dataReceived(self, data):
        stdout.write(data)

class EchoClientFactory(ClientFactory):

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

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

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

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

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

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

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

Reconnection

Many times, the connection of a client will be lost unintentionally due to network errors. 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)
```

3.4.4 A Higher-Level Example: ircLogBot

Overview of ircLogBot

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

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

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

  <foo> logbot: hello!

the bot will reply:

  <logbot> foo: I am a log bot

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

  $ python ircLogBot.py test test.log

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


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

# system imports
import time, sys


class MessageLogger:
    """
    An independant 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."""

    def __init__(self):
        self.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))
        if msg.startswith("%s:" % self.nickname):
            # someone is talking to me, lets respond:
            msg = "%s: I am a log bot" % user
            self.say(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))


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

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

    # the class of the protocol to build when new connection is made
    protocol = LogBot

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

    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 to.

```
from twisted.internet import protocol
from twisted.protocols import irc
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):

    protocol = LogBot

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

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.

3.5 UDP Networking

3.5.1 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 multiple connections, we only use a single object, a protocol, for each UDP socket. We then use the reactor to connect this protocol to a UDP transport, using the `twisted.internet.interfaces.IReactorUDP` reactor API.

3.5.2 DatagramProtocol

At the base, the place where you actually implement the protocol parsing and handling, is the DatagramProtocol class. This class will usually be decended from `twisted.internet.protocol.DatagramProtocol`. Most protocol handlers inherit either from this class 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, and when sending datagrams the address to send to 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 registed with the reactor. This means it may be persisted if it's added to an application, and thus it has `twisted.internet.protocol.DatagramProtocol.startProtocol` and `twisted.internet.protocol.DatagramProtocol.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, not a hostname. If you only have the hostname use `reactor.resolve()` to resolve the address (see `twisted.internet.interfaces.IReactorCore.resolve`).

3.5.3 Connected UDP

A connected UDP socket is slighly 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 is faster.

Unlike a regular UDP protocol, we do not need to specify where to send datagrams to, and are not told where they came from since they can only come from address the socket is 'connected' to.

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

class Helloer(DatagramProtocol):
```

```
    def startProtocol(self):
        d = self.transport.connect("192.168.1.1", 1234)
        d.addCallback(self._cbConnected)

    def _cbConnected(self, (host, port)):
        print "we can only send to %s now" % str((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)

# 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 domain names. To obtain the IP of a domain name use `reactor.resolve()`, e.g.:

```
from twisted.internet import reactor

def gotIP(ip):
print "IP of 'example.com' is", ip

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

Connecting to a new address after a previous connection, or making a connected port unconnected are not currently supported, but will likely be supported in the future.

3.6 Using Processes

3.6.1 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.protocol.ProcessProtocol`

3.6.2 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 six more: arguments, environment, path, userID, groupID, and usePTY.

```
from twisted.internet import reactor

mypp = 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. Interactive programs (where you don't know when it may read or write) need to be run with ptys.
- `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 the `twisted.internet.interfaces.IProcessTransport`.

3.6.3 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 the 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 receieved in dataReceived, data from standard output is receieve 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()
```

3.6.4 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.
- `.processEnded(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. This scheme may seem a bit weird, but I trust that it proves useful when dealing with exceptions that occur in asynchronous code.

  This will always be called *after*`inConnectionLost`, `outConnectionLost`, and `errConnectionLost` are called.

The base-class definitions of these functions are all 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.

3.6.5 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.

- `os.kill(self.transport.pid, signal.SIGKILL)`: Kill the child process. This will eventually result in `processEnded` being called.

3.6.6 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/python

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 processEnded(self, status_object):
        print "processEnded, status %d" % status_object.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.
%
```

3.6.7 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*

3.6.8 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 `.dataReceived` 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.
- `os.kill(self.transport.pid, signal.SIGKILL)`: Kill the child process. This will eventually result in `processEnded` being called.

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).

3.7 Using Deferreds

3.7.1 Introduction

Twisted is a framework that allows programmers to develop *asynchronous* (page 96) networked programs. `twisted.internet.defer.Deferred` objects are one of the key concepts that you should understand in order to develop asynchronous code that uses the Twisted framework: they are a signal to the calling function that a result is pending.

This HOWTO first describes the problem that Deferreds solve: that of managing tasks that are waiting for data without blocking. It then illustrates the difference between blocking Python code, and non-blocking code which returns a Deferred; describes Deferreds in more details; describes the details of the class interfaces; and finally describes DeferredList.

3.7.2 The Problem that Deferreds Solve

Deferreds are designed to enable Twisted programs to wait for data without hanging until that data arrives.

Many computing tasks take some time to complete, and there are two reasons why a task might take some time:

1. it is computationally intensive (for example factorising large numbers) and requires a certain amount of CPU time to calculate the answer; or
2. it is not computationally intensive but has to wait for data to be available to produce a result.

It is the second class of problem — non-computationally intensive tasks that involve an appreciable delay — that Deferreds are designed to help solve. Functions that wait on hard drive access, database access, and network access all fall into this class, although the time delay varies.

The basic idea behind Deferreds, and other solutions to this problem, is to keep the CPU as active as possible. If one task is waiting on data, rather than have the CPU (and the program!) idle waiting for that data (a process normally called ”blocking”), the program performs other operations in the meantime, and waits for some signal that data is ready to be processed before returning to that process.

In Twisted, a function signals to the calling function that it is waiting by returning a Deferred. When the data is available, the program activates the callbacks on that Deferred to process the data.

3.7.3 The Context

Dealing with Blocking Code

When coding I/O based programs - networking code, databases, file access - there are many APIs that are blocking, and many methods where the common idiom is to block until a result is gotten.

```
class Getter:
    def getData(self, x):
        # imagine I/O blocking code here
        print "blocking"
        import time
        time.sleep(4)
        return x * 3

g = Getter()
print g.getData(3)
```

Don’t Call Us, We’ll Call You

Twisted cannot support blocking calls in most of its code, since it is single threaded, and event based. The solution for this issue is to refactor the code, so that instead of blocking until data is available, we return immediately, and use a callback to notify the requester once the data eventually arrives.

```
from twisted.internet import reactor

class Getter:
    def getData(self, x, callback):
        # this won’t block
        reactor.callLater(2, callback, x * 3)

def printData(d):
    print d

g = Getter()
g.getData(3, printData)

# startup the event loop, exiting after 4 seconds
reactor.callLater(4, reactor.stop);
reactor.run()
```

There are several things missing in this simple example. There is no way to know if the data never comes back; no mechanism for handling errors. The example does not handle multiple callback functions, nor does it give a method to merge arguments before and after execution. Further, there is no way to distinguish between different calls to gotData from different producer objects. `Deferred` solves these problems, by creating a single, unified way to handle callbacks and errors from deferred execution.

3.7.4 Deferreds

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

class Getter:
    def getData(self, x):
        # this won't block
        d = defer.Deferred()
        reactor.callLater(2, d.callback, x * 3)
        return d

def printData(d):
    print d

g = Getter()
d = g.getData(3)
d.addCallback(printData)

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

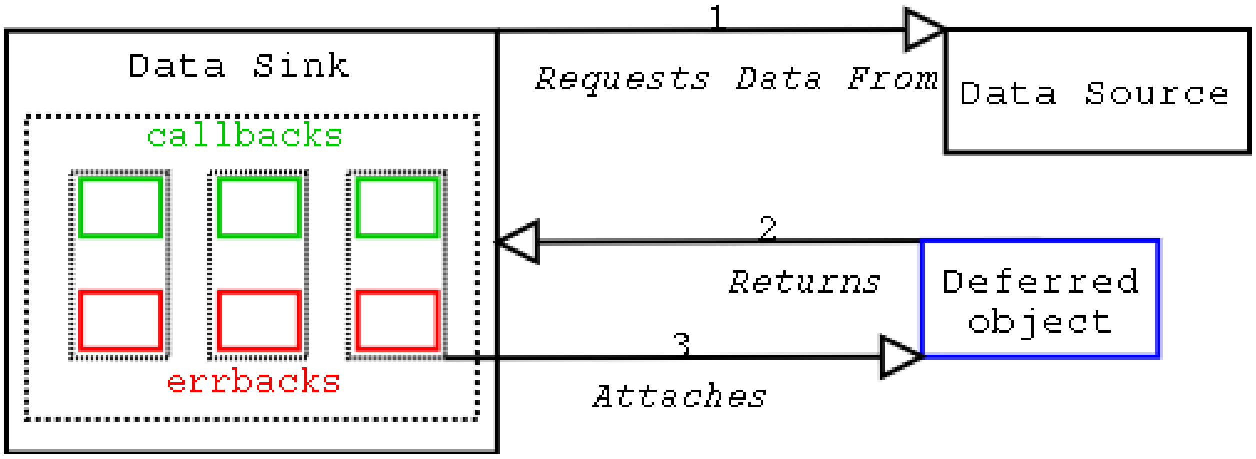
Deferreds do not make the code magically not block. Once you have rewritten your code to not block, Deferreds give you a nice way to build an interface to that code.

As we said, 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.
        """
        if x % 2 == 0:
            self.d.callback(x*3)
        else:
```

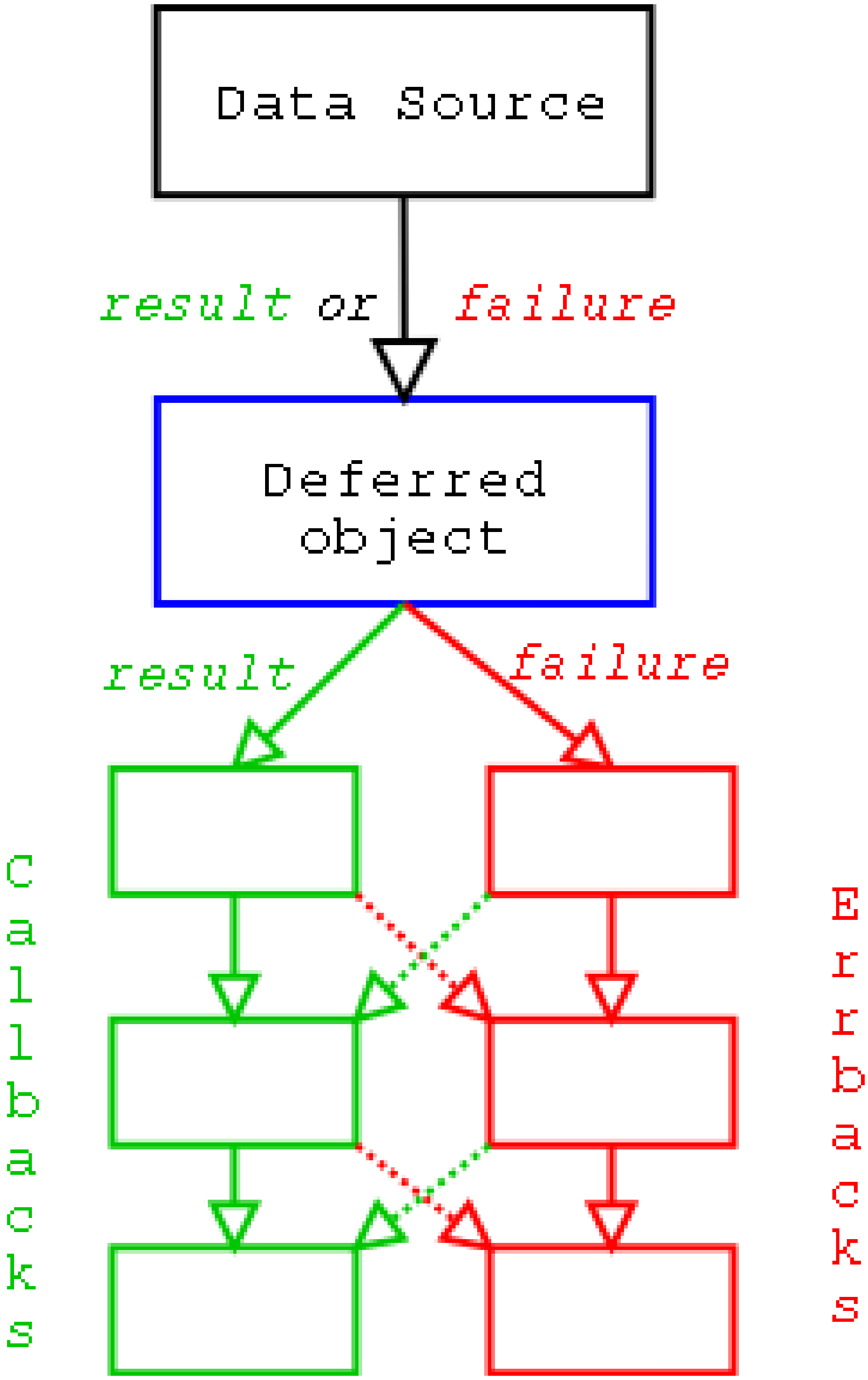
```
            self.d.errback(ValueError("You used an odd number!"))

    def _toHTML(self, r):
        return "Result: %s" % r

    def getData(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.
        """
        self.d = defer.Deferred()
        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 will print an error message
g = Getter()
d = g.getData(3)
d.addCallback(printData)
d.addErrback(printError)

# this will print "Result: 12"
g = Getter()
d = g.getData(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.

More about callbacks

You add multiple callbacks to a Deferred:

```
g = Getter()
d = g.getResult(3)
d.addCallback(processResult)
d.addCallback(printResult)
```

Each callback feeds its return value into the next callback (callbacks will be called in the order you add them). Thus in the previous example, `processResult`'s return value will be passed to `printResult`, instead of the value initially passed into the callback. This gives you a flexible way to chain results together, possibly modifying values along the way (for example, you may wish to pre-process database query results).

More about 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, "A" will handle a success condition from `getDeferredFromSomewhere`, and "B" will handle any errors that occur *from either the upstream source, or that occur in 'A'*. In Case 2, "C"'s errback1 *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)
```

3.7.5 Class Overview

This is the overview API reference for Deferred. It is not meant to be a substitute for the docstrings in the Deferred class, but can provide guidelines for its use.

Basic Callback Functions

- `addCallbacks(self, callback[, errback, callbackArgs, errbackArgs, errbackKeywords, asDefaults])`

  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.

  - `addCallback(callback, *callbackArgs, **callbackKeywords)`

    Adds your callback at the next point in the processing chain, while adding an errback that will re-raise its first argument, not affecting further processing in the error case.

    Note that, while addCallbacks (plural) requires the arguments to be passed in a tuple, addCallback (singular) takes all its remaining arguments as things to be passed to the callback function. The reason is obvious: addCallbacks (plural) cannot tell whether the arguments are meant for the callback or the errback, so they must be specifically marked by putting them into a tuple. addCallback (singular) knows that everything is destined to go to the callback, so it can use Python's "*" and "**" syntax to collect the remaining arguments.

  - `addErrback(errback, *errbackArgs, **errbackKeywords)`

    Adds your errback at the next point in the processing chain, while adding a callback that will return its first argument, not affecting further processing in the success case.

  - `addBoth(callbackOrErrback, *callbackOrErrbackArgs, **callbackOrErrbackKeywords)`

    This method adds the same callback into both sides of the processing chain at both points. Keep in mind that the type of the first argument is indeterminate if you use this method! Use it for `finally:` style blocks.

- `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.

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)`

Automatic Error Conditions

- `setTimeout(seconds[, timeoutFunc])`

  Set a timeout function to be triggered if this Deferred is not called within that time period. By default, this will raise a TimeoutError after `seconds`.

A Brief Interlude: Technical Details

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.

Deferreds are one-shot. A generalization of the Deferred API to generic event-sources is in progress – watch this space for updates! – but Deferred itself is only for events that occur once. 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.

The important consequence of this is that *sometimes, addCallbacks will call its argument synchronously, and sometimes it will not.* In situations where callbacks modify state, it is highly 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; code that calls `callback` or `errback` should *never* call `unpause`, as this would negate its usefulness!

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.

3.7.6 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(d, user):
    return user in ["Alice", "Angus", "Agnes"]
```

However, here's an `asynchronousIsValidUser` function that returns a Deferred:

```
from twisted.internet import reactor

def asynchronousIsValidUser(d, user):
    d = 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`.

Returning a Deferred from synchronous functions

An alternative is for `authenticateUser` to require that the implementation of `isValidUser` return a Deferred:

```
def printResult(result):
    if result:
        print "User is authenticated"
    else:
        print "User is not authenticated"

def authenticateUser(isValidUser, user):
    d = isValidUser(user)
    d.addCallback(printResult)
```

In this case, the author of `synchronousIsValidUser` would use `defer.succeed` to return a fired Deferred which will call the first callback with `result`, rather than returning the result itself:

```
from twisted.internet import defer

def immediateIsValidUser(d, user):
    result = user in ["Alice", "Angus", "Agnes"]
    return defer.succeed(result)
```

3.7.7 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:

```
def printResult(result):
    print result
deferred1 = defer.Deferred()
deferred2 = defer.Deferred()
deferred3 = defer.Deferred()
dl = defer.DeferredList([deferred1, deferred2, deferred3])
dl.addCallback(printResult)
deferred1.callback('one')
deferred2.errback('bang!')
deferred3.callback('three')
# At this point, dl will fire its callback, printing:
#     [(1, 'one'), (0, 'bang!'), (1, 'three')]
# (note that defer.SUCCESS == 1, and defer.FAILURE == 0)
```

A standard DeferredList will never call errback.

> **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 two keywords 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].

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")
```

If we want a task to run every X seconds repeatedly, we can use `twisted.internet.task.LoopingCall`:

[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.

```
from twisted.internet import task

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
```

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()
```

3.9 Using Threads in Twisted

3.9.1 Introduction

Before you start using threads, make sure you do at the start of your program:

```
from twisted.python import threadable
threadable.init()
```

This will make certain parts of Twisted thread-safe so you can use them safely. However, note that most parts of Twisted are *not* thread-safe.

3.9.2 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
from twisted.python import threadable
threadable.init(1)

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
```

3.9.3 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")
```

3.9.4 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 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)
```

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

```
from twisted.internet import 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)
```

3.9.5 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 Choosing a Reactor and GUI Toolkit Integration

3.10.1 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 general purpose reactor implementations are:

- *The select()-based reactor* (page 137)
- *The poll()-based reactor* (page 138)

Platform-specific reactor implementations exist for:

- *cReactor for Unix* (page 138)
- *KQueue for FreeBSD* (page 138)
- *Win32* (page 138)
- *Mac OS X* (page 139)

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+ 1.2 and 2.0* (page 138)
- *Qt* (page 139)
- *Tkinter* (page 139)
- *WxPython* (page 139)
- *Win32* (page 138)
- *Cocoa* (page 139)
- *PyUI* (page 140)

When using applications that runnable using `twistd`, e.g. TAPs 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 hasen't already installed one.

3.10.2 Reactor Functionality

	TCP	**SSL**	**UDP**	**Threading**	**Processes**	**Scheduling**	**Platforms**
select()	Y	Y	Y	Y	Y (Unix only)	Y	Unix, Win32
poll()	Y	Y	Y	Y	Y	Y	Unix
Win32	Y	Y	Y	Y	Y	Y	Win32
cfreactor	Y	Y	Y	Y	Y	Y	OS X
GTK+	Y	Y	Y	Y	Y (Unix only)	Y	Unix, Win32
Qt	Y	Y	Y	Y	Y (Unix only)	Y	Unix, Win32
kqueue	Y	Y	Y	Y	Y	Y	FreeBSD
C	Y	N	N	Y	Y	Y	Unix

Table 3.1: Summary of reactor features

3.10.3 General Purpose Reactors

Select()-based Reactor

The SelectReactor is the default reactor.

```
from twisted.internet import reactor
```

The SelectReactor may be explicitly installed by:

```
from twisted.internet import default
default.install()
```

Poll()-based Reactor

The PollReactor will work on any platform that provides `poll()`. With larger numbers of connected sockets, it may provide for better performance.

```
from twisted.internet import pollreactor
pollreactor.install()
```

3.10.4 Platform-Specific Reactors

cReactor for Unix

The cReactor is a high-performance C implementation of the Reactor interfaces. It is currently experimental and under active development.

```
from twisted.internet import cReactor
cReactor.install()
```

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()
```

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()
```

3.10.5 GUI Integration Reactors

GTK+

Twisted integrates with PyGTK[4], versions 1.2 and 2.0. Sample applications using GTK+ and Twisted are available in the Twisted CVS.

```
from twisted.internet import gtkreactor
gtkreactor.install()
```

[4] http://www.daa.com.au/~james/pygtk/

3.10.6 GUI Integration Reactors

Cocoa

Twisted integrates with PyObjC[5], version 1.0. Sample applications using Cocoa and Twisted are available in the examples directory under `Cocoa`.

```
from twisted.internet import cfreactor
cfreactor.install()
```

Qt

An example Twisted application that uses Qt can be found in `doc/examples/qtdemo.py`.

When installing the reactor, pass a QApplication instance, and if you don't a new one will be created for you.

```
from qt import QApplication
app = QApplication([])

from twisted.internet import qtreactor
qtreactor.install(app)
```

3.10.7 Non-Reactor GUI Integration

Tkinter

The support for Tkinter[6] doesn't use a specialized reactor. Instead, there is some specialized support code:

```
from Tkinter import *
from twisted.internet import tksupport

root = Tk()
root.withdraw()

# 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()".
```

wxPython

As with *Tkinter* (this page), the support for integrating Twisted with a wxPython[7] application uses specialized support code rather than a simple reactor.

---

[5] http://pyobjc.sf.net/
[6] http://www.python.org/topics/tkinter/
[7] http://www.wxpython.org

```
from wxPython.wx import *
from twisted.internet import wxsupport, reactor

myWxAppInstance = wxApp(0)
wxsupport.install(myWxAppInstance)
```

However, this has issues when runnin 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()
```

Later, once a `wxApp` instance has been created, but before `reactor.run()` is called:

```
myWxAppInstance = wxApp(0)
reactor.registerWxApp(myWxAppInstance)
```

An example Twisted application that uses WxWindows can be found in `doc/examples/wxdemo.py`.

PyUI

As with *Tkinter* (page 139), the support for integrating Twisted with a PyUI[8] 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 bve found in `doc/examples/pyuidemo.py`.

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

Chapter 4

High-Level Twisted

4.1 The Basics

4.1.1 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 assortment of tools to create and manipulate `Applications`.

`Applications` are just Python objects, which can be created and manipulated in the same ways as any other object. In particular, they can be serialized to files. Twisted supports several serialization formats.

4.1.2 Serialization

TAP A Twisted Application Pickle. This format is supported by the native Python pickle support. While not being human readable, this format is the fastest to load and save.

TAX Twisted contains `twisted.persisted.marmalade`, a module that supports serializing and deserializing from a format which follows the XML standard. This format is human readable and editable.

TAS Twisted contains `twisted.persisted.aot`, a module that supports serializing into Python source. This has the advantage of using Python's own parser and being able to later manually add Python code to the file.

4.1.3 mktap and tapconvert

The `mktap(1)` utility is the main way to create a TAP (or TAX or TAS) file. It can be used to create an Application for all of the major Twisted server types like web, ftp or IRC. It also supports plugins, so when you install a Twisted plugin (that is, unpack it into a directory on your `PYTHONPATH`) it will automatically detect it and use any Twisted Application support in it. It can create any of the above Application formats.

In order to see which server types are available, use `mktap --help`. For a given server, `mktap --help <name>` shows the possible configuration options. `mktap` supports a number of generic options to configure the application – for full details, read the man page.

One important option is `--append <filename>`. This is used when there is already a Twisted application serialized to which a server should be added. For example, it can be used to add a telnet server, which would let you probe and reconfigure the application by telnetting into it.

Another useful utility is `tapconvert(1)`, which converts between all three Application formats.

4.1.4 twistd

Having an `Application` in a variety of formats, aesthetically pleasing as it may be, does not actually cause anything to happen. For that, we need a program which takes a "dead" Application and brings life to it. For UNIX systems (and, until there are are alternatives, for other operating systems too), this program is `twistd(1)`. Strictly speaking, `twistd` is not necessary – unserializing 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 136)), logging to a logfile, daemonizing and more. `twistd` supports all Applications mentioned above – and an additional one. Sometimes is 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 twisted.pid` ``). When the process is killed in an orderly fashion it will leave behind the "shutdown Application" which is named the same as the original file with a `-shutdown` added to its base name. This contains the new configuration information, as changed in the application. For example, `web.tap` when shutdown will have an additional file, `web-shutdown.tap`.

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

4.1.5 tap2deb

For Twisted-based server application developers who want to deploy on Debian, Twisted supplies the `tap2deb` program. This program wraps a Twisted Application file (of any of the supported formats – Python, source, xml or pickle) in a Debian package, including correct installation and removal scripts and `init.d` scripts. This frees the installer from manually stopping or starting the service, and will make sure it goes properly up on startup and down on shutdown and that it obeys the init levels.

For the more savvy Debian users, the `tap2deb` also generates the source package, allowing her to modify and polish things which automated software cannot detect (such as dependencies or relationships to virtual packages). In addition, the Twisted team itself intends to produce Debian packages for some common services, such as web servers and an inetd replacement. Those packages will enjoy the best of all worlds – both the consistency which comes from being based on the `tap2deb` and the delicate manual tweaking of a Debian maintainer, insuring perfect integration with Debian.

Right now, there is a beta Debian archive of a web server available at Moshe's archive[1].

[1] http://twistedmatrix.com/users/moshez/apt

4.1.6 tap2rpm

`tap2rpm` is similar to `tap2deb`, except that it generates RPMs for Redhat and other related platforms.

4.2 Twisted 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 110 Volt socket, and one of them a foreign 220 Volt socket. If we plug the hair dryer into the 220 Volt socket, it is going to expect 110 Volt current and errors will result. Going back and changing the hair dryer to support both `plug110Volt` and `plug220Volt` 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() == 110:
            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 110

class ForeignSocket:
    def voltage(self):
        return 220
```

Given these classes, the following operations can be performed:

```
>>> hd = HairDryer()
>>> am = AmericanSocket()
>>> hd.plug(am)
I was plugged in properly and am operating.
>>> fs = ForeignSocket()
>>> hd.plug(fs)
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 `ForeignSocket` 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. There is a simple implementation in `twisted.python.components.Adapter` which defines the `__init__` shown below, so you can subclass it if you desire. 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()
>>> fs = ForeignSocket()
>>> adapted = AdaptToAmericanSocket(fs)
>>> 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 twisted.python.components: Twisted's implementation of Interfaces and Components

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 `twisted.python.components.Interface`, and have a very odd look to python programmers not used to them:

```
from twisted.python import components

class IAmericanSocket(components.Interface):
    def voltage(self):
      """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 `components.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 `ForeignSocket` 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:

```
class AmericanSocket:
    __implements__ = (IAmericanSocket, )
    def voltage(self):
        return 110
```

So, to declare that a class implements an interface, we simply set the `__implements__` class variable to a tuple of interfaces. A single item tuple in Python is created by enclosing an item in parentheses and placing a single trailing comma after it.

Now, let's say we want to rewrite the `AdaptToAmericanSocket` class as a real adapter. We simply subclass `components.Adapter` and provide implementations of the methods in the `IAmericanSocket` interface:

```
class AdaptToAmericanSocket(components.Adapter):
    __implements__ = (IAmericanSocket, )
    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 `ForeignSocket` object, we can *register `AdaptToAmericanSocket` as an `IAmericanSocket` adapter for the `ForeignSocket` class*. It is easier to see how this is done in code than to describe it:

```
from twisted.python import components

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

class AmericanSocket:
    __implements__ = (IAmericanSocket, )
    def voltage(self):
        return 110

class ForeignSocket:
    def voltage(self):
        return 220

class AdaptToAmericanSocket(components.Adapter):
    __implements__ = (IAmericanSocket, )
    def voltage(self):
        return self.original.voltage() / 2

components.registerAdapter(
    AdaptToAmericanSocket,
    ForeignSocket,
    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:

```
>>> as = AmericanSocket()
>>> fs = ForeignSocket()
>>> components.implements(as, IAmericanSocket)
1
>>> components.implements(fs, IAmericanSocket)
0
```

As you can see, the `AmericanSocket` instance claims to implement `IAmericanSocket`, but the `Foreign Socket` 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 `ForeignSocket` instance, we must *adapt* it to `IAmericanSocket` before doing so. We use the interface object to do this:

```
>>> IAmericanSocket(fs)
<__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 `ForeignSocket`. But what happens if we attempt to adapt an object which already implements `IAmericanSocket`? We simply get back the original instance:

```
>>> IAmericanSocket(as)
<__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 socket.voltage() == 110, "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:

```
>>> as = AmericanSocket()
>>> fs = ForeignSocket()
>>> hd = HairDryer()
>>> hd.plug(as)
I was plugged in properly and am operating
>>> hd.plug(fs)
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 defines its own `__implements__` tuple (because it implements an additional interface), you must explicitly include the base clase interface list. The Interface code does not walk the complete inheritance tree (because that would be a significant performance hit), but instead will recursively expand any tuples it finds in each class's `__implements__` attribute.

For example, `pb.Root` (actually defined in `flavors.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__` attribute like:

```
class Root(Referenceable):
    __implements__ = IPBRoot,
```

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

```
class MyThing:
    __implements__ = IMyInterface,
```

Now if you want to make this class inherit from `pb.Root`, you must manually include `pb.Root`'s interface list:

```
class MyThing(pb.Root):
    __implements__ = (IMyInterface, pb.Root.__implements__)
```

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.

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 `ICredentialChecker` which resolves some Credentials to an avatar ID. Some examples of CredentialChecker implementations would be: InMemoryUsernamePassword, ApacheStyleHTAccessFile, UNIXPasswordDatabase, SSHPublicKeyDatabase. 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 twisted.protocols import basic
from twisted.python import log, components
from twisted.cred import credentials, error
from twisted.internet import defer

class IMailbox(components.Interface):
    """Interface specification for mailbox."""
```

```
    def deleteMessage(self, 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()
```

4.4 Using Application

4.4.1 Motivation

What happens when you want to write a Twisted application? Well, if you come into Twisted thinking of it as a library, you may decide a useful way is to write a Python script. This Python script would probably take the following actions:

- Install the reactor you want (or skip this step if you want the default reactor).
- Perform initialization options – open data files, start `reactor.callLater` loops, etc.
- Call `reactor.run()`
- Perform deinitalization options

There are two problems with this methodology. It has a lot of boiler-plate code, and it introduces an inflexibility into the design. The usual way to solve this kind of problem is to write *configuration files*, and it is no different in Twisted.

At this point, the standard thing to do would be to write a long, tedious and subtly wrong manual describing the configuration language. Rest assured, like every other project, Twisted has those. But the language is secondary, and will be described later – more important are the configuration *objects*.

4.4.2 Services, Persistence and Security, Oh My!

It is possible to think of any configuration language as a specially designed language to build a configuration object, which is then queried and acted upon by the program runtime. In Twisted, this is literally true internally – and the master configuration object is `twisted.application.service.Application`. However, there is virtually nothing you can do with this object directly. This object is `Componentized` – it has different, orthogonal, aspects. Access to this aspects is done by using *interfaces*. Interfaces, for our purposes, are just callables which return different aspects of the `Application`. Unlike other frameworks, like `Qt` or `wxWindows`, in Twisted you do not derive from `Application` – you use methods to register your objects with it.

There are four interfaces supported, three of which are defined in `twisted.application.service`:

- `IService`
- `IServiceCollection`
- `IProcess`
- `twisted.persisted.sob.IPersistable`

Constructing an application is done by calling it with a single argument – its name. The name influences some things, among them the default file it will persist to (which is why it is mandatory).

```
from twisted.application import service
from twisted.persisted import sob
application = service.Application("myapplication")
s = service.IService(application)
sc = service.IServiceCollection(application)
proc = service.IProcess(application)
per = sob.IPersistable(application)
```

4.4.3 Services

There are two interfaces relevant to services – `IService` and `IServiceCollection`. `IService` represents a *state-aware* container. That means the service is ready to be notified of application start-ups and shutdowns. Services can be named or unnamed. `IServiceCollection` holds other services. It is possible to get named services from it by name. All services can be gotten from it via either indexing or iteration.

Services can have a parent. Parents are set using `setServiceParent`. Services are detached from their parent with `disownServiceParent`. The parent must always be something that complies with the `IServiceCollection` interface.

Most services will inherit from `Service`. This class will set an attribute `running` to a true value in `startService` and to a false value in `stopService`. This attribute will always be false in just-unpersisted `Services`, without regards to its value at the time the `Service` was persisted.

`MultiService` implements both `IService` and `IServiceCollection`. It is used to keep the services in a hierarchy.

It is, of course, possible to write one's own services, but Twisted comes out of the box with several services which are useful in writing network applications. These are found in `twisted.application.internet`, including `TCPServer`, `TCPClient`, and `TimerService`.

To each `reactor.listenFoo` method corresponds a service named `FooServer`. The arguments to its constructor are the same as the arguments to the method. It calls the method on application start-up, and stops listening on application shut-down.

To each `reactor.connectFoo` methods corresponds a service named `FooClient`. The arguments to its constructor are the same as the arguments to the method. It calls the method on application start-up. It might, or might not, stop the connection on application shut-down. (This limitation will be removed at some point, and guaranteed disconnection will be implemented.)

The last service in `twisted.application.internet` is `TimerService`. The constructor takes a period, a callable and optionally arguments and keyword arguments. The service, when it is running, will make sure the callable will be called every time the period elapses.

4.4.4 String Ports

In Twisted, a `ServerFactory` does not care what kind of virtual reliable circuit it listens too – SSL, UNIX domain sockets, TCP sockets or something else. However, the APIs for constructing the various `*Server` classes are different. When it is necessary for a less sophisticated user to direct construction of such a class, the `twisted.application.strports` module comes in handy. It contains a function `service` which accepts a *description* and a factory, and returns a service. The description is a string in a mini-language designed to specify ports. Full specifications are in the module docstrings.

4.4.5 Configuration

At some point, the objects for the configuration actually have to be constructed. The easiest and simplest way to do it is to use Python as the configuration mini-language. This format is called `TAC` and traditionally files with this format have the extension `.tac`.

TAC files need to be valid Python files, which construct a variable named `application`. This variable will be the configuration object. The full power of Python is available, of course.

Here's an example:

```
# Import modules
from twisted.application import service, internet
from twisted.protocols import wire
from twisted.internet import protocol

# Construct the application
application = service.Application("echo")

# Get the IServiceCollection interface
myService = service.IServiceCollection(application)

# Create the protocol factory
myFactory = protocol.ServerFactory()
myFactory.protocol = wire.Echo

# Create the (sole) server
# Normally, the echo protocol lives on port 7, but since that
# is a privileged port, for this example we'll use port 7001
myServer = internet.TCPServer(7001, myFactory)

# Tie the service to the application
myServer.setServiceParent(myService)
```

Note that `setServiceParent` will, in fact, automatically cast its argument to `IServiceCollection`. So, more succinctly, the above code can be written:

```
from twisted.application import service, internet
from twisted.protocols import wire
from twisted.internet import protocol
application = service.Application("echo")
myFactory = protocol.ServerFactory()
myFactory.protocol = wire.Echo
internet.TCPServer(7001, myFactory).setServiceParent(application)
```

TAC files are run with `twistd -y` or `twistd --python`. The `twistd` manpage has more information, but a common way to run is `twistd -noy echo.tac`. This tells `twistd` to not daemonize and not to try and save application on shutdown.

4.5 Writing a New Plug-In for mktap

4.5.1 Getting Started

Twisted is a very general and powerful tool. It can power anything connected to a network, from your corporate message-broadcasting network to your desktop IRC client. This is great for integrating lots of different tools, but can make it very difficult to document and understand how the whole platform is supposed to work. A side effect of this is that it's hard to get started with a project using Twisted, because it's hard to find out where to start.

This guide is to help you understand one of the main ways to get started working on a Twisted server application. It probably won't answer your specific questions about how to do things like *schedule functions to call in the future* (page 133) or *listen on a socket* (page 98); there are other documents that address these concerns and you can read them later. Client applications are not likely to use the infrastructure described in this document, and a simpler alternative exists for servers, covered in the *Application howto* (page 154).

4.5.2 Twisted and You: Where Does Your Code Fit In?

If you're like most people that have asked me questions about this, you've probably come to Twisted thinking of it as a library of code to help you write an application. It can be, but it is much more useful to think of *your code as the library*. Twisted is a framework.

The difference between a framework and a library is that a developer's code will run a library's functions; a framework runs the developer's functions, instead. The difference is subtle, but significant; there are a range of resources which have to be allocated and managed regarding start-up and shut-down of an process, such as spawning of threads and handling events. You don't have to use Twisted this way. It is quite possible to write applications that use Twisted almost exclusively as a library. If you use it as a framework, though, Twisted will help you by managing these resources itself.

The central framework classes that you will deal with, both as a Twisted developer and administrator, are services, which implement `twisted.application.service.IService`. `twisted.application.service.Application` creates the root of a tree of services that form a twisted application. There is one `Application` instance per Twisted process, and it is the top-level manager of resources and handler of events in the Twisted framework. (Unlike some other frameworks, developers do not subclass `Application`; rather by adding services it, you register event handlers to be called by it.) To store configuration data, as well as other information, Twisted serializes `Application` instances, storing all services that have been registered with them. Since the whole `Application` instance is serialized, Twisted "configuration" files are significantly more comprehensive than those for other systems. These files store everything related to a running `Application` instance; in essence the full state of a running process.

The central concept that a Twisted system administrator will work with are files that contain `Application` instances serialized in various formats optimized for different uses. `.TAP` files are optimized for speed of loading and saving, `.TAX` files are editable by administrators familiar with XML syntax, and `.TAS` files are generated Python source code, most useful for developers. The two command-line programs which work with these files are `mktap` and `twistd`. The `mktap` utility create `.TA*` files from simple command-line arguments, and the `twistd` daemon will load and run those files. Alternatively, a Python script can be used to create the `Application` instance and this script can be run directly using `twistd -y script.py`, as long as the file `script.py` has a `Application` object called `application` on the module level. Applications are covered in more depth in the *Application howto* (page 154).

There are many ways in which your code will be called by various parts of the Twisted framework by the time you're done. The initial one we're going to focus on here is a plug-in for the `mktap` utility. `mktap` produces complete, runnable `Application` instances, so no additional work is necessary to make your code work with `twistd`. First we will go through the process of creating a plug-in that Twisted can find, then we make it adhere to the mktap interface. Finally we will load that plug-in with a server.

4.5.3 What is a Plug-In?

Python makes it very easy to dynamically load and evaluate programs. The plug-in system for Twisted, twisted.python.plugin, is a way to find (without loading) and then load plug-ins for particular systems.

Unlike other "plug-in" systems, like the well known ones associated with The Gimp, Photoshop, and Apache twisted.python.plugin is generic. Any one of the Twisted "dot-products"[2] can define mechanisms for extensibility using plug-ins. Two Twisted dot-products already load such plug-ins. The twisted.applicationpackage loads Twisted Application builder modules (TAP plug-ins) and the twisted.lore package loads document formatting modules.

Twisted finds its plug-ins by using pre-existing Python concepts; the load path, and packages. Every top-level Python package[3] (that is, a directory whose parent is on sys.path and which contains an __init__.py) can potentially contain some number of plug-ins. Packages which contain plug-ins are called "drop-ins", because you "drop" them into your sys.path. The only difference between a package and a drop-in is the existence of a file named plugins.tml (TML for Twisted Module List) that contains some special Python expressions to identify the location of sub-packages or modules which can be loaded.

If you look at twisted/plugins.tml, you will notice that Twisted is a drop-in for itself! You can browse through it for lots of examples of plug-ins being registered.

The most prevalent kind of plug-in is the TAP (Twisted Application builder) type. These are relatively simple to get started with. Let's look at an excerpt from Twisted's own plugins.tml for an example of registering one:

```
# ...

register("Twisted Web Automated TAP builder",
         "twisted.tap.web",
         description="""
         Builds a Twisted Application instance that contains a general-purpose
         web server, which can serve from a filesystem or application resource.
         """,
         type="tap",
         tapname="web")

# ...
```

plugins.tml will be a list of calls to one function:

```
register(name, module, type=plugin_type,
         description=user_description
         [, **plugin_specific_data])
```

- name is a free-form string, to be displayed to the user in presentation contexts (like a web page, or a list-box in a GUI).

[2] http://twistedmatrix.com/products/dot-products
[3] http://www.python.org/doc/current/tut/node8.html#SECTION008400000000000000000

- `module` is a string which must be the fully-qualified name of a Python module.
- `type` is the name of the system you are plugging in to. Be sure to spell this right, or Twisted won't find your plug-in at all!
- `**plugin_specific_data` is a dictionary of information associated with the plug-in, specific to the `type` of plug-in it is. Note that some plug-in types may require a specific bit of data in order to work.

Note the `tapname` parameter given in the example above. This parameter is an example of `**plugin_specific_data`. The parameter `tapname` is only used by `"tap"`-type modules. It indicates what name to use on the `mktap` command line. In English, this particular call to `register` means "When the user types `mktap web`, it selects the module `twisted.tap.web` to handle the rest of the arguments".

Now that you understand how to register a plug-in, let's move along to writing your first one.

4.5.4 Twisted Quotes: A Case Study

As an example, we are going to work on a Quote of the Day application, `TwistedQuotes`. Aspects of this application will be explored in more depth throughout in the Twisted documentation.

`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.

Before you Begin

First, make a directory, `TwistedQuotes`, where you're going to keep your code. If you installed Twisted from source, the path of least resistance is probably just to make a directory inside your `Twisted-X.X.X` directory, which will already be in your `sys.path`. If you want to put it elsewhere, make sure that your `TwistedQuotes` directory is a package on your python path.

> **Note:**
> The directory you add to your PYTHONPATH needs to be the directory *containing* your package's directory! For example, if your TwistedQuotes directory is /my/stuff/TwistedQuotes, you can `export PYTHONPATH=/my/stuff:$PYTHONPATH` in UNIX, or edit the `PYTHONPATH` environment variable to add `/my/stuff;` at the beginning through the System Properties dialog on Windows.

You will then need to add an `__init__.py` to this directory, to mark it as a package. (For more information on exactly how Python packages work, read this section[4] of the Python tutorial.) In order to test that everything is working, start up the Python interactive interpreter, or your favorite IDE, and verify that the package imports properly.

```
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.
```

[4] http://www.python.org/doc/current/tut/node8.html#SECTION008400000000000000000

A Look at the Heart of the Application

(You'll need to put this code into a file called `quoters.py` in your `TwistedQuotes` directory.)

```
from twisted.python import components

from random import choice


class IQuoter(components.Interface):
    """An object that returns quotes."""

    def getQuote(self):
        """Return a quote."""


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

    __implements__ = IQuoter

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

    def getQuote(self):
        return self.quote


class FortuneQuoter:
    """Load quotes from a fortune-format file."""

    __implements__ = IQuoter

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

    def getQuote(self):
        return choice(open(choice(self.filenames)).read().split('\n%\n'))
```

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 120). (Don't worry we'll get to each of those features in later documentation.)

Your Twisted applications should follow this style as much as possible. Have (at least) one module which implements your specific functionality, independant 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.

(You'll need to put this code into a file called `quoteproto.py` in your `TwistedQuotes` directory.)

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

class QOTD(Protocol):

    def connectionMade(self):
        self.transport.write(self.factory.quoter.getQuote()+'\r\n')
        self.transport.loseConnection()

class QOTDFactory(Factory):

    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 the `mktap` plug-in below.

Note: you can read more specifics of `Protocol` and `Factory` in the *Writing Servers* (page 98) 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.

Practically speaking, this link is an interface for a savvy user who will run the server. (In this case, you; when you have more users, a system administrator.) For the purposes of this example we will first implement a `mktap` interface.

Like most system administrator tools, this is command-line oriented. (It is possible to implement a graphical front-end to mktap, using the same plug-in structure, but this has not been done yet.)

Creating the extension to `mktap` is done through implementing a module that follows the `mktap` plug-in interface, and then registering it to be found and loaded by `twisted.python.plugin`. As described above, registration is done by adding a call to `register` in the file `TwistedQuotes/plugins.tml`

(You'll need to put this code into a file called `quotetap.py` in your `TwistedQuotes` directory.)

```
from twisted.application import internet # services that run TCP/SSL/etc.
from TwistedQuotes import quoteproto    # Protocol and Factory
from TwistedQuotes import quoters       # "give me a quote" code

from twisted.python import usage        # twisted command-line processing


class Options(usage.Options):
    optParameters = [["port", "p", 8007,
                      "Port number to listen on for QOTD protocol."],
                     ["static", "s", "An apple a day keeps the doctor away.",
                      "A static quote to display."],
                     ["file", "f", None,
                      "A fortune-format text file to read quotes from."]]


def makeService(config):
    """Return a service that will be attached to the application."""
    if config["file"]:                  # If I was given a "file" option...
        # Read quotes from a file, selecting a random one each time,
        quoter = quoters.FortuneQuoter([config['file']])
    else:                               # otherwise,
        # read a single quote from the command line (or use the default).
        quoter = quoters.StaticQuoter(config['static'])
    port = int(config["port"])          # TCP port to listen on
    factory = quoteproto.QOTDFactory(quoter) # here we create a QOTDFactory
    # Finally, set up our factory, with its custom quoter, to create QOTD
    # protocol instances when events arrive on the specified port.
    return internet.TCPServer(port, factory)
```

Twisted Quotes TAP construction module — *quotetap.py*

This module has to conform to a fairly simple interface. It must have a class called `Options` which is a subclass of `twisted.python.usage.Options`. It must also have a function `makeService(config)`, which will be passed an instance of the `Options` class defined in the module itself, `TwistedQuotes.quotetap.Options`. Command-line options given on the `mktap` command line fill in the values in `Options` and are used in `makeService` to make the actual connections between objects. `makeService` is expected to return an object implementing `IService`. This can be a `Service` subclass, a `MultiService` collection of sub-services, a `TCPServer` serving a protocol factory, and so on.

A more detailed discussion of `twisted.python.usage.Options` can be found in the document *Using `usage.Options`* (page 181).

Now that we've implemented all the necessary pieces, we can finish putting them together by writing a TML file which allows the `mktap` utility to find our protocol module.

```
register("Quote of the Day TAP Builder",
         "TwistedQuotes.quotetap",
         description="""
         Example of a TAP builder module.
         """,
         type="tap",
         tapname="qotd")
```

Twisted Quotes Plug-in registration — *plugins.tml*

Now the QOTD server is ready to be instantiated! Let's start up a server and get a quote from it.

```
% mktap qotd
Saving qotd application to qotd.tap...
Saved.
% twistd -f qotd.tap
% nc localhost 8007
An apple a day keeps the doctor away.
% kill `cat twistd.pid`
```

Let's walk through the above example. First, we run `mktap` specifying the Application type (`qotd`) to create. `mktap` reads in our `plugins.tml` file, instantiates an `Application` object, fills in the appropriate data, and serializes it out to a `qotd.tap` file. Next, we launch the server using the twistd daemon, passing `qotd.tap` as a command line option. The server launches, listens on the default port from `quotetap.py`. Next, we run `nc` to connect to the running server. In this step, the `QOTDFactory` creates a `Quoter` instance, which responds to our network connection by sending a quote string (in this case, the default quote) over our connection, and then closes the connection. Finally, we shutdown the server by killing it via a saved out process id file.

(`nc` is the netcat[5] utility, which no UNIX system should be without.)

So we just saw Twisted in action as a framework. With relatively little code, we've got a server that can respond to a request over a network, with two potential alternative back-ends (fortune files and static text).

After reading this (and following along with your own example, of course), you should be familiar with the process of getting your own Twisted code with unique functionality in it running inside of a server. You should be familiar with the concept of a drop-in and a plug-in, and understand both how to create them and how to install them from other people on your system.

By following the rules set out at the beginning of this HOWTO, we have accidentally implemented another piece of useful functionality.

```
% mktap
Usage:    mktap [options] <command> [command options]
```

[5] http://www.atstake.com/research/tools/index.html#network_utilities

```
Options:
  -x, --xml      DEPRECATED: same as --type=xml
  -s, --source   DEPRECATED: same as --type=source
  -e, --encrypted  Encrypt file before writing
  -p, --progress   Show progress of plugin loading
  -d, --debug      Show debug information for plugin loading
  -u, --uid=     [default: 1000]
  -g, --gid=     [default: 1000]
  -a, --append=  An existing .tap file to append the plugin to, rather than
                 creating a new one.
  -t, --type=    The output format to use; this can be 'pickle', 'xml', or
                 'source'. [default: pickle]
      --help     display this message
Commands:
    ftp              An FTP server.
    im               A multi-protocol chat client.
    inetd
    mail             An email service.
    manhole          An interactive remote debugger service.
    news             News Server
    portforward      A simple port-forwarder.
    qotd             Example of a TAP builder module.
    socks            A SOCKSv4 proxy service.
    ssh
    telnet           A simple, telnet-based remote debugging service.
    toc              An AIM TOC service.
    web              A general-purpose web server which can serve from a
                     filesystem or application resource.
    words            A chat service.
```

Not only does our `Options` class get instantiated by `mktap` directly, the user can query mktap for interactive help! This is just one small benefit to using Twisted as it was designed. As more tools that use the `tap` style of plug-in, more useful functionality will become available from Twisted Quotes. For example, a graphical tool could provide not just help messages at the command line, but a listing of all available TAP types and forms for each, for the user to enter information.

It is this kind of power that results from using a dynamic, powerful framework like Twisted. I hope that you take your newfound knowledge and discover all kinds of cool things like this that you get for free just by using it!

The plug-in system is a relatively new part of Twisted, and not as many things use it as they should yet. Watch this space for new developments regarding plug-ins, other systems that you can plug your code into, and more documentation for people wanting to write systems that can be plugged in to!

4.6 Upgrading Applications

Applications must frequently deal with data that lives longer than the programs that create it. Sometimes the structure of that data changes over time, but new versions of a program must be able to accomodate data created by an older version. These versions may change very quickly, especially during development of new code. Sometimes different versions of the same program are running at the same time, sharing data across a network connection. These situations all result in a need for a way to upgrade data structures.

4.6.1 Basic Persistence: Application and .tap files

Simple object persistence (using `pickle` or `jelly`) provides the fundamental "save the object to disk" functionality at application shutdown. If you use the `Application` object, every object referenced by your Application will be saved into the `-shutdown.tap` file when the program terminates. When you use `twistd` to launch that new .tap file, the Application object will be restored along with all of its referenced data.

This provides a simple way to have data outlive any particular invocation of your program: simply store it as an attribute of the Application. Note that all Services are referenced by the Application, so their attributes will be stored as well. Ports that have been bound with listenTCP (and the like) are also remembered, and the sockets are created at startup time (when `Application.run` is called).

To influence the way that the `Application` is persisted, you can adapt it to `twisted.persisted.sob.IPersistable` and use the `setStyle(style)` method with a string like "pickle", "xml", or "source". These use different serializers (and different extensions: ".tap", ".tax", and ".tas" respectively) for the saved Application.

You can manually cause the application to be saved by calling its `.save` method (on the `twisted.persisted.sob.IPersistable` adapted object).

4.6.2 Versioned: New Code Meets Old Data

So suppose you're running version 1 of some application, and you want to upgrade to version 2. You shut down the program, giving you a .tap file that you could restore with twistd to get back to the same state that you had before. The upgrade process is to then install the new version of the application, and then use twistd to launch the saved .tap file. The old data will be loaded into classes created with the new code, and now you'll have a program running with the new behavior but the old data.

But what about the data structures that have changed? Since these structures are really just pickled class instances, the real question is what about the class definitions that have changed? Changes to class methods are easy: nothing about them is saved in the .tap file. The issue is when the data attributes of a instance are added, removed, or their format is changed.

Twisted provides a mechanism called `Versioned` to ease these upgrades. Each version of the data structure (i.e. each version of the class) gets a version number. This number must change every time you add or remove a data attribute to the class. It must also change every time you modify one of those data attributes: for example, if you use a string in one version and an integer in another, those versions must have different version numbers.

The version number is defined in a class attribute named `persistenceVersion`. This is an integer which will be stored in the .tap file along with the rest of the instance state. When the object is unserialized, the saved persistenceVersion is compared against the current class's value, and if they differ, special upgrade methods are called. These methods are named `upgradeToVersionNN`, and there must be one for each intermediate version. These methods are expected to manipulate the instance's state from the previous version's format into that of the new version.

To use this, simply have your class inherit from `Versioned`. You don't have to do this from the very beginning of time: all objects have an implicit version number of "0" when they don't inherit from Versioned. So when you first make an incompatible data-format change to your class, add Versioned to the inheritance list, and add an `upgradeToVersion1` method.

For example, suppose the first version of our class saves an integer which measures the size of a line. We release this as version 1.0 of our neat application:

```
class Thing:
  def __init__(self, length):
      self.length = length
```

Then we fix some bugs elsewhere, and release versions 1.1 and 1.2 of the application. Later, we decide that we should add some units to the length, so that people can refer to it in inches or meters. Version 1.3 is shipped with the following code:

```
class Thing(Versioned):
  persistenceVersion = 1
  def __init__(self, length, units):
      self.length = "%d %s" % (length, units)
  def upgradeToVersion1(self):
      self.length = "%d inches" % self.length
```

Note that we must make an assumption about what the previous value meant: in this case, we assume the number was in inches.

1.4 and 1.5 are shipped with other changes. Then in version 1.6 we decide that saving the two values as a string was foolish and that it would be better to save the number and the string separately, using a tuple. We ship 1.6 with the following:

```
class Thing(Versioned):
  persistenceVersion = 2
  def __init__(self, length, units):
      self.length = (length, units)
  def upgradeToVersion1(self):
      self.length = "%d inches" % self.length
  def upgradeToVersion2(self):
      (length, units) = self.length.split()
      self.length = (length, units)
```

Note that we must provide both `upgradeToVersion1`*and*`upgradeToVersion2`. We have to assume that the saved .tap files which will be provided to this class come from a random assortment of old versions: we must be prepared to accept anything ever saved by a released version of our application.

Finally, version 2.0 adds multiple dimensions. Instead of merely recording the length of a line, it records the size of an N-dimensional rectangular solid. For backwards compatiblity, all 1.X version of the program are assumed to be dealing with a 1-dimensional line. We change the name of the attribute from `.length` to `.size` to reflect the new meaning.

```
class Thing(Versioned):
  persistenceVersion = 3
  def __init__(self, dimensions):
      # dimensions is a list of tuples, each is (length, units)
      self.size = dimensions
      self.name = ["line", "square", "cube", "hypercube"][len(dimensions)]
  def upgradeToVersion1(self):
      self.length = "%d inches" % self.length
  def upgradeToVersion2(self):
```

```
        (length, units) = self.length.split()
        self.length = (length, units)
    def upgradeToVersion3(self):
        self.size = [self.length]
        del self.length
        self.name = "line"
```

If a .tap file from the earliest version of our program were to be loaded by the latest code, the following sequence would occur for each Thing instance contained inside:

1. An instance of Thing would be created, with a __dict__ that contained a single attribute `.size`, which was an integer, like "5".
2. `self.upgradeToVersion1()` would be called, changing `self.size` into a string, like "5 inches".
3. `self.upgradeToVersion2()` would be called, changing `self.size` into a tuple, like (5, "inches").
4. Finally, `self.upgradeToVersion3()` would be called, creating `self.size` as a list holding a single dimension, like [(5, "inches")]. The old `.length` attribute is deleted, and a new `.name` is created with the type of shape this instance represents ("line").

Some hints for the `upgradeVersion` methods:

- They must do everything the `__init__` method would have done, as well as any methods that might have been called during the lifetime of the object.
- If the class has (or used to have) methods which can add attributes that weren't created in `__init__`, then the saved object may have a haphazard subset of those attributes, depending upon which methods were called. The upgradeVersion methods must be prepared to deal with this. `hasattr` and `.get` may be useful.
- Once you have released a class with a given `upgradeVersion` method, you should never change that method. (assuming you care about infinite backwards compatibility).
- You must add a new `upgradeVersion` method (and bump the persistenceVersion value) for each and every release that has a different set of data attributes than the previous release.
- `Versioned` works by providing `__setstate__` and `__getstate__` methods. You probably don't want to override these methods without being very careful to call the Versioned versions at exactly the right time. It also requires a `doUpgrade` function to be called after all the objects are loaded. This is done automatically by `Application.run`.
- Depending upon how they are serialized, `Versioned` objects can probably be sent across a network connection, and the upgrade process can be made to occur upon receipt. (You'll want to look at the `requireUpgrade` function). This might be useful in providing compability with an older peer. Note, however, that `Versioned` does not let you go backwards in time; there is no `downgradeVersionNN` method. This means it is probably only useful for compatibility in one direction: the newer-to-older direction must still be explicitly handled by the application.
- In general, backwards compatibility is handled by pretending that the old code was restricting itself to a narrow subset of the capabilities of the new code. The job of the upgrade routines is then to translate the old representation into a new one.

For more information, look at the doc strings for `styles.Versioned`, as well as the `app.Application` class and the *Application HOWTO* (page 154).

4.6.3 Rebuild: Loading New Code Without Restarting

`Versioned` is good for handling changes between released versions of your program, where the application state is saved on disk during the upgrade. But while you are developing that code, you often want to change the behavior of the running program, *without* the slowdown of saving everything out to disk, shutting down, and restarting. Sometimes it will be difficult or time-consuming to get back to the previous state: the running program could include ephemeral objects (like open sockets) which cannot be persisted.

`twisted.python.rebuild` provides a function called `rebuild` which helps smooth this cycle. It allows objects in a running program to be upgraded to a new version of the code without shutting down.

To use it, simply call `rebuild` on the module that holds the classes you want to be upgraded. Through deep `gc` magic, all instances of classes in that module will be located and upgraded.

Typically, this is done in response to a privileged command sent over a network connection. The usual development cycle is to start the server, get it into an interesting state, see a problem, edit the class definition, then push the "rebuild yourself" button. That "button" could be a magic web page which, when requested, runs `rebuild(mymodule)`, or a special IRC command, or perhaps just a socket that listens for connections and accepts a password to trigger the rebuild. (You want this to be a privileged operation to prevent someone from making your server do a rebuild while you're in the middle of editing the code).

A few useful notes about the rebuild process:

- If the module has a top-level attribute named `ALLOW_TWISTED_REBUILD`, this attribute must evaluate to True. Should it be false, the rebuild attempt will raise an exception.
- Adapters (from `twisted.python.components`) use top-level registration function calls. These are handled correctly during rebuilds, and the usual duplicate registration errors are not raised.
- Rebuilds may be slow: every single object known to the interpreter must be examined to see if it is one of the classes being changed.

Finally, note that `rebuild`*cannot* currently be mixed with `Versioned`. `rebuild` does not run any of the classes' methods, whereas `Versioned` works by running `__setstate__` during the load process and `doUpgrade` afterwards. This means `rebuild` can only be used to process upgrades that do not change the data attributes of any of the involved classes. Any time attributes are added or removed, the program must be shut down, persisted, and restarted, with upgradeToVersionNN methods used to handle the attributes. (this may change in the future, but for now the implementation is easier and more reliable with this restriction).

4.7 Cooperative Data Flows (via generators)

4.7.1 Background

Handling more than one Request

When creating services to handle simultaneous requests, processing state must be managed. A common approach, threading, applies the method used for single user operations: keeping processing state within local variables and function arguments. In this case, the operating environment handles variable allocation and cleanup. Unfortunately, threading requires much preemption and locking magic to keep each thread's execution independent, yet allowing data to be shared between threads. This magic can lead to subtle programming errors, and, in the case of Python, slowness as every changes to a mutable object must be synchronized with the global interpreter lock.

An alternative is to use a event-driven approach, where each operation is broken down into distinct steps, each step scheduled for operation in an delayed execution queue. In this approach, an object is often used used to maintain state information between each step, allowing other operations to be performed. Often each step is a member function, queued within Twisted using `reactor.CallLater`.

```
from twisted.internet import reactor

class Request:
    def __init__(self, name):
        self.name = name
        reactor.callLater(0, self.step_one)
    def step_one(self):
        print self.name, ": Step One"
        reactor.callLater(0, self.step_two)
    def step_two(self):
        print self.name, ": Step Two"
        reactor.callLater(0, self.step_done)
    def step_done(self):
        # cleanup, and don't call reactor again.
        pass

Request("James")
Request("Wendy")

# start and shut down the event loop
reactor.callLater(1, reactor.stop)
reactor.run()
```

In the output of this example, one can see that execution alternates between steps for James and Wendy's request. In effect, the requests are done in a parallel, cooperative manner. While this is a good start, it isn't perfect as the event loop is stopped with a hard-coded timeout. It would be better for each Request object to signal when it is completed so that the event loop can be shut down sooner.

Deferred Callbacks

In Twisted, communication between operations is often accomplished with `defer.Deferred`. The Deferred object decouples an asyncronous operation by providing a temporary storage location for its result. Instead of calling the operation and waiting for a return value, an operation schedules itself to be executed with `reactor.callLater`and then returns a `defer.Deferred` object. At this point, the caller can register a callback function to be executed when the operation has completed. In the example below, `defer.DeferredList` is used to merge the result of each request into a single notification, which is used to stop the main event loop.

```
from twisted.internet import reactor, defer
```

```
class Request:
    def __init__(self, name):
        self.name = name
        self.d    = defer.Deferred()
        reactor.callLater(0, self.step_one)
    def step_one(self):
        print self.name, ": Step One"
        reactor.callLater(0, self.step_two)
    def step_two(self):
        print self.name, ": Step Two"
        reactor.callLater(0, self.step_done)
    def step_done(self):
        self.d.callback(None) # notify callback that we are done

james = Request("James")
wendy = Request("Wendy")

# start and shut down the event loop
d = defer.DeferredList([james.d, wendy.d])
d.addCallback(lambda _: reactor.stop())
reactor.run()
```

While this `Deferred` approach is very good, it can get quickly complicated, especially if the request is not a simple linear sequence of steps, or when results must flow between steps incrementally. The flow module addresses these shortcomings using python generators.

4.7.2 Iterators and generators

An iterator is basically an object which produces a sequence of values. Python's iterators are simply objects with an `__iter__()` member function which returns an object (usually itself) which has a `next()` member function. The `next()` method is then invoked till it raises a `StopIteration` exception.

```
from twisted.python.compat import iter, StopIteration

class Counter:
    def __init__(self, count):
        self.count = count
    def __iter__(self):
        return self
    def next(self):
        ret = self.count
        self.count -= 1
        if ret: return ret
        raise StopIteration
        return ret

import sys
```

```
if sys.version_info < (2,2):
    def list(it):
        ret = []
        it = iter(it)
        try:
            while 1:
                ret.append(it.next())
        except StopIteration: pass
        return ret

print list(Counter(3))

# prints: [3, 2, 1]
```

State pattern

Often times it is useful for an iterator to change state during its production of values. This can be done nicely with the 'state' pattern.

```
class States:
    def __iter__(self):
        self.next = self.state_one
        return self
    def state_one(self):
        self.next = self.state_two
        return "one"
    def state_two(self):
        self.next = self.state_stop
        return "two"
    def state_stop(self):
        raise StopIteration

print list(States())

# prints: ['one', 'two']
```

Generators

With Python 2.2, there is a wonderful syntax sugar for creating iterators... generators. When a generator is first executed, an iterator is returned. And from there on, each invocation of `next()` gives the subsequent value produced by the `yield` statement. With generators, the two iterators above become very easy to express.

```
from __future__ import generators

def Counter(count):
    while count > 0:
        yield count
```

```
        count -= 1

def States():
    yield "one"
    yield "two"

print list(Counter(3))
print list(States())

# prints:
#     [3, 2, 1]
#     ['one', 'two']
```

One technical difference between iterators and generators, is that raising an exception from a generator permanently halts the generator, while raising an exception from an iterator's `next()` method does not invalidate the iterator, that is, one could call the `next()` method again and possibly get results. From here on, we use the generator syntax for building iterators.

Chaining Generators

It is often useful to view an operation information as a flow between stages, where each stage may have several states or steps. This can be coded where the output of one generator is consumed by another. In this view, the last generator in the chain 'pulls' data from previous stages.

```
from __future__ import generators

def Counter(count):
    while count > 0:
        yield count
        count -= 1

def Consumer():
    producer = Counter(3)
    for result in producer:
        if 2 != result:
            yield result

print list(Consumer())

# prints: [3, 1]
```

While this is a very clean syntax for creating a multi-stage operation, it would block all other operations. Therefore, some mechanism for pausing the generator and resuming it later is required.

4.7.3 Introducing Flow

The flow module provides this ability to cooperate with other tasks. This is accomplished by wrapping iterables in a flow stage object and following an alternating yield pattern. That is, before each value pulled from the stage, the operation must yield the wrapper object. During this yield bookkeeping is done to prepare the next value, or, if the next value is not available, re-scheduling the operation to be executed later.

```
from __future__ import generators
from twisted.flow import flow

def Counter(count):
    while count > 0:
        yield count
        count -= 1

def Consumer():
    producer = flow.wrap(Counter(3))
    yield producer
    for result in producer:
        if 2 != result:
            yield result
        yield producer

print list(flow.Block(Consumer))

# prints: [3, 1]
```

Equivalent Forms

In the above code, `producer.next()` is called implicitly, and thus the generator above is equivalent to...

```
from __future__ import generators
def Consumer():
    producer = flow.wrap(Counter(3))
    while True:
        yield producer
        result = producer.next()
        if 2 != result:
            yield result
```

The `next()` method of the wrapper object does several things. First, it checks to see if there are results ready, if so it returns the next one. If not, it looks for a failure, raising it. And finally, checking to see if the end of the input has been reached. More concretely...

```
from __future__ import generators
def Consumer():
    producer = flow.wrap(Counter(3))
    while True:
        yield producer
        if producer.results:
            result = producer.results.pop(0)
            yield result
```

```
            continue
        if producer.failure:
            producer.stop = 1
            producer.failure.trap()
        if producer.stop:
            break
```

Handling failures

Another difference between plain old iterables and one wrapped with the flow module is that exceptions caught are wrapped with a `twisted.python.failure.Failure` object for later delivery. There are two basic ways to recover from exceptions. One way is to list expected exceptions in the call to `flow.wrap`. Alternatively, a try/except block can be used, catching `flow.Failure` objects.

```
from __future__ import generators
from twisted.flow import flow

def Producer(throw):
    yield 1
    yield 2
    raise throw
    yield 3

def Consumer(producer):
    producer = flow.wrap(producer, IOError)
    yield producer
    try:
        for result in producer:
            if result is IOError:
                # handle trapped error
                yield "trapped"
            else:
                yield result
            yield producer
    except AssertionError, err:
        # handle assertion error
        yield str(err)

print list(flow.Block(Consumer(Producer(IOError("trap")))))
print list(flow.Block(Consumer(Producer(AssertionError("notrap")))))

# prints: [1, 2, 'trapped']
# prints: [1, 2, 'notrap']
```

Cooperate

This seems like quite the effort, wrapping each iterator and then having to alter the calling sequence. Why? The answer is that it allows for a `flow.Cooperate` object to be returned. When this happens, the entire call chain can be paused so that other flows can use the call stack. For flow.Block, the implementation of Cooperate simply puts the call chain to sleep.

```
from __future__ import generators
from twisted.flow import flow

def gen():
    yield 'immediate'
    yield flow.Cooperate(2)
    yield 'delayed'

for x in flow.Block(gen):
   print x

# prints:
#  immediate
#  delayed
```

Merge and Zip

Cooperate can be demonstrated with `flow.Merge` and `flow.Zip` components. These two stages join two or more generators into a single stream. The Merge operation does this by rotating between any of its input streams which are available. The Zip operation, on the other hand, waits for a result from each stream before it produces a result.

```
from __future__ import generators
from twisted.flow import flow

def Right():
    yield "one"
    yield "two"
    yield flow.Cooperate()
    yield "three"

def Left():
    yield 1
    yield 2
    yield 3

print "Zip", list(flow.Block(flow.Zip(Right,Left)))
print "Merge", list(flow.Block(flow.Merge(Right,Left)))

# Zip [('one', 1), ('two', 2), ('three', 3)]
# Merge ['one', 1, 'two', 2, 3, 'three']
```

4.7.4 Integrating with Twisted

While `flow.Block` is useful for understanding how flow works, it undermines the whole purpose by sleeping during Cooperate and blocking other operations. Following are numerous examples of how to integrate flow with the Twisted framework in a non-blocking manner.

Deferred Flow

For starters, the long example of Wendy and James, with its numerous calls to `reactor.callLater` to schedule each step of the operation can be rewritten using flow.Deferred.

```
from __future__ import generators
from twisted.internet import reactor, defer
from twisted.flow import flow

def request(name):
    print name, ": Step One"
    yield flow.Cooperate()
    print name, ": Step Two"

james = flow.Deferred(request("James"))
wendy = flow.Deferred(request("Wendy"))

# start and shut down the event loop
d = defer.DeferredList([wendy, james])
d.addCallback(lambda _: reactor.stop())
reactor.run()
```

Under the sheets, when `flow.Deferred` encounters a `flow.Cooperate` event, it reschedules itself to be resumed at a later time, allowing other asyncronous operations to proceed. Once again, `defer.DeferredList` is only used here to stop the reactor after all operations are completed.

Flow Resources

By using `flow.Deferred` it is easy to make up a web resource which is both long running, but also can serve more than one customer at a time. Run the example below, and with two browsers, view the webpage. Notice that both web pages are being created at the same time.

```
from __future__ import generators
from twisted.internet import reactor
from twisted.web import server, resource
from twisted.flow import flow

def cooperative(count):
    """ simulate a cooperative resource, that not block """
    from random import random
    idx = 0
    while idx < count:
```

```
        val = random()
        yield flow.Cooperate(val)
        yield str(val)[-5:]
        idx += 1

def flowRender(req):
    count = int(req.args.get("count",["30"])[0])
    req.write("<html><body>")
    req.write(" %s Random numbers: <ol>\n" % count)
    source = flow.wrap(cooperative(count))
    yield source
    for itm in source:
        req.write("<li>%s</li>\n" % itm)
        yield source
    req.write("</ol></body></html>\n")

class FlowResource(resource.Resource):
    def __init__(self, gen):
        resource.Resource.__init__(self)
        self.gen = gen
    def isLeaf(self): return true
    def render(self, req):
        self.d = flow.Deferred(self.gen(req))
        self.d.addCallback(lambda _: req.finish())
        return server.NOT_DONE_YET

print "visit http://localhost:8081/ to view the example"
root = FlowResource(flowRender)
site = server.Site(root)
reactor.listenTCP(8081,site)
reactor.run()
```

Flow Protocols

The flow module can also be used to construct protocols easily, following is an echo client and server. For each protocol, one must yield the connection before reading from it. When the generator finishes, the connection is automatically closed.

```
from __future__ import generators
from twisted.flow import flow
from twisted.internet import protocol, reactor
PORT = 8392

def echoServer(conn):
    yield conn
    for data in conn:
```

```
        conn.write(data)
        yield conn
    reactor.callLater(0,reactor.stop)

server = protocol.ServerFactory()
server.protocol = flow.makeProtocol(echoServer)
reactor.listenTCP(PORT,server)

def echoClient(conn):
    conn.write("Hello World")
    yield conn
    print conn.next()
    conn.write("Another Line")
    yield conn
    print conn.next()

client = protocol.ClientFactory()
client.protocol = flow.makeProtocol(echoClient)
reactor.connectTCP("localhost", PORT, client)
reactor.run()
```

4.7.5 Threading

While the Flow module allows for multiple cooperative tasks to work in a single thread, sometimes it is necessary to have the output of another thread be consumed within a flow. This can be done with `twisted.flow.threads.Threaded`, which takes an iterable object and executes it in another thread. Following is a sample iterable, countSleep which simulates a blocking producer which must be put into a thread. To show that `twisted.flow.threads.Threaded` does not block other operations, a similar, cooperative count is included.

```
from __future__ import generators
from twisted.internet import reactor, defer
from twisted.flow import flow
from twisted.flow.threads import Threaded

def countSleep(index):
    from time import sleep
    for index in range(index):
        sleep(.3)
        print "sleep", index
        yield index

def countCooperate(index):
    for index in range(index):
        yield flow.Cooperate(.1)
        print "cooperate", index
        yield "coop %s" % index
```

```
d = flow.Deferred( flow.Merge(
        Threaded(countSleep(5)),
        countCooperate(5)))

# # alternatively
# d1 = flow.Deferred(Threaded(countSleep(5)))
# d2 = flow.Deferred(countCooperate(10))
# d = defer.DeferredList([d1,d2])

def prn(x):
    print x
    reactor.stop()
d.addCallback(prn)
reactor.run()
```

Using database connections

Since most standard database drivers are thread based, the flow builds on the `twisted.flow.threads.Threaded` by providing a `QueryIterator`, which takes an sql query and a `ConnectionPool`.

```
from __future__          import generators
from twisted.enterprise import adbapi
from twisted.internet   import reactor
from twisted.flow import flow
from twisted.flow.threads import QueryIterator, Threaded

dbpool = adbapi.ConnectionPool("SomeDriver",host='localhost',
             db='Database',user='User',passwd='Password')

# # I test with...
# from pyPgSQL import PgSQL
# dbpool = PgSQL

sql = """
  (SELECT 'one')
UNION ALL
  (SELECT 'two')
UNION ALL
  (SELECT 'three')
"""
def consumer():
    print "executing"
    query = Threaded(QueryIterator(dbpool, sql))
    print "yielding"
    yield query
```

```
    print "done yeilding"
    for row in query:
        print "Processed result : ", row
        yield query

from twisted.internet import reactor
def finish(result):
    print "Deferred Complete : ", result
    reactor.stop()
f = flow.Deferred(consumer())
f.addBoth(finish)
reactor.run()
```

Chapter 5

Utilities

5.1 Using usage.Options

5.1.1 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
```

5.1.2 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"]
    ]
```

5.1.3 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.

5.1.4 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.

5.1.5 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.

5.1.6 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)`.

5.1.7 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"
```

5.2 Twisted RDBMS support

5.2.1 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.

5.2.2 What you should already know

- Python :-)
- How to write a simple Twisted Server (see *this tutorial* (page 98) to learn how)
- Familiarity with using database interfaces (see the documentation for DBAPI 2.0[1] or this article[2] by Andrew Kuchling)

[1] http://www.python.org/topics/database/DatabaseAPI-2.0.html
[2] http://www.amk.ca/python/writing/DB-API.html

5.2.3 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[3]-compliant module.

`enterprise.adbapi` will do blocking database operations in seperate 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.

5.2.4 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"
```

[3] http://www.python.org/topics/database/DatabaseAPI-2.0.html

```
    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 120).

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.

```
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, ...)`.

5.2.5 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.

5.3 Twisted Enterprise Row Objects

The `twisted.enterprise.row` module is a method of interfacing simple python objects with rows in relational database tables. It has two components: the `RowObject` class which developers sub-class for each relational table that their code interacts with, and the `Reflector` which is responsible for updates, inserts, queries and deletes against the database.

The row module is intended for applications such as on-line games, and websites that require a back-end database interface. It is not a full functioned object-relational mapper for python - it deals best with simple data types structured in ways that can be easily represented in a relational database. It is well suited to building a python interface to an existing relational database, and slightly less suited to added database persistance to an existing python application.

If row does not fit your model, you will be best off using the low-level database API (page 186) directly, or writing your own object/relational layer on top of it.

5.3.1 Class Definitions

To interface to relational database tables, the developer must create a class derived from the `twisted.enterprise.row.RowObject` class for each table. These derived classes must define a number of class attributes which contains information about the database table that class corresponds to. The required class attributes are:

- rowColumns - list of the column names and types in the table with the correct case
- rowKeyColumns - list of key columns in form: `[(columnName, typeName)]`
- rowTableName - the name of the database table

There are also two optional class attributes that can be specified:

- rowForeignKeys - list of foreign keys to other database tables in the form: `[(tableName, [(childColumnName, childColumnType), ...], [(parentColumnName, parentColumnType), ...], containerMethodName, autoLoad]`
- rowFactoryMethod - a method that creates instances of this class

For example:

```
class RoomRow(row.RowObject):
    rowColumns       = [("roomId",  "int"),
                         ("town_id", "int"),
                         ("name",    "varchar"),
                         ("owner",   "varchar"),
                         ("posx",    "int"),
                         ("posy",    "int"),
                         ("width",   "int"),
                         ("height",  "int")]
    rowKeyColumns    = [("roomId",  "int4")]
    rowTableName     = "testrooms"
    rowFactoryMethod = [testRoomFactory]
```

The items in the rowColumns list will become data members of classes of this type when they are created by the Reflector.

5.3.2 Initialization

The initialization phase builds the SQL for the database interactions. It uses the system catalogs of the database to do this, but requires some basic information to get started. The class attributes of the classes derived from RowClass are used for this. Those classes are passed to a Reflector when it is created.

There are currently two available reflectors in Twisted Enterprise, the SQL Reflector for relational databases which uses the python DB API, and the XML Reflector which uses a file system containing XML files. The XML reflector is currently extremely slow.

An example class list for the RoomRow class we specified above using the SQLReflector:

```
from twisted.enterprise.sqlreflector import SQLReflector

dbpool = adbapi.ConnectionPool("pyPgSQL.PgSQL")
reflector = SQLReflector( dbpool, [RoomRow] )
```

5.3.3 Creating Row Objects

There are two methods of creating RowObjects - loading from the database, and creating a new instance ready to be inserted.

To load rows from the database and create RowObject instances for each of the rows, use the loadObjectsFrom method of the Reflector. This takes a tableName, an optional "user data" parameter, and an optional "where clause". The where clause may be omitted which will retrieve all the rows from the table. For example:

```
def gotRooms(rooms):
    for room in rooms:
        print "Got room:", room.id

d = reflector.loadObjectsFrom("testrooms",
                              whereClause=[("id", reflector.EQUAL, 5)])
d.addCallback(gotRooms)
```

For more advanced RowObject construction, loadObjectsFrom may use a factoryMethod that was specified as a class attribute for the RowClass derived class. This method will be called for each of the rows with the class object, the userData parameter, and a dictionary of data from the database keyed by column name. This factory method should return a fully populated RowObject instance and may be used to do pre-processing, lookups, and data transformations before exposing the data to user code. An example factory method:

```
def testRoomFactory(roomClass, userData, kw):
    newRoom = roomClass(userData)
    newRoom.__dict__.update(kw)
    return newRoom
```

The last method of creating a row object is for new instances that do not already exist in the database table. In this case, create a new instance and assign its primary key attributes and all of its member data attributes, then pass it to the `insertRow` method of the Reflector. For example:

```
    newRoom = RoomRow()
    newRoom.assignKeyAttr("roomI", 11)
```

```
newRoom.town_id = 20
newRoom.name = 'newRoom1'
newRoom.owner = 'fred'
newRoom.posx = 100
newRoom.posy = 100
newRoom.width = 15
newRoom.height = 20
reflector.insertRow(newRoom).addCallback(onInsert)
```

This will insert a new row into the database table for this new RowObject instance. Note that the `assignKeyAttr` method must be used to set primary key attributes - regular attribute assignment of a primary key attribute of a rowObject will raise an exception. This prevents the database identity of RowObject from being changed by mistake.

5.3.4 Relationships Between Tables

Specifying a foreign key for a RowClass creates a relationship between database tables. When `loadObjectsFrom` is called for a table, it will automatically load all the children rows for the rows from the specified table. The child rows will be put into a list member variable of the rowObject instance with the name `childRows` or if a *containerMethod* is specified for the foreign key relationship, that method will be called on the parent row object for each row that is being added to it as a child.

The *autoLoad* member of the foreign key definition is a flag that specifies whether child rows should be auto-loaded for that relationship when a parent row is loaded.

5.3.5 Duplicate Row Objects

If a reflector tries to load an instance of a rowObject that is already loaded, it will return a reference to the existing rowObject rather than creating a new instance. The reflector maintains a cache of weak references to all loaded row objects by their unique keys for this purpose.

5.3.6 Updating Row Objects

RowObjects have a `dirty` member attribute that is set to 1 when any of the member attributes of the instance that map to database columns are changed. This dirty flag can be used to tell when RowObjects need to be updated back to the database. In addition, the `setDirty` method can be overridden to provide more complex automated handling such as dirty lists (be sure to call the base class setDirty though!).

When it is determined that a RowObject instance is dirty and need to have its state updated into the database, pass that object to the `updateRow` method of the Reflector. For example:

```
reflector.updateRow(room).addCallback(onUpdated)
```

For more complex behavior, the reflector can generate the SQL for the update but not perform the update. This can be useful for batching up multiple updates into single requests. For example:

```
updateSQL = reflector.updateRowSQL(room)
```

5.3.7 Deleting Row Objects

To delete a row from a database pass the RowObject instance for that row to the Reflector `deleteRow` method. Deleting the python Rowobject instance does *not* automatically delete the row from the database. For example:

```
reflector.deleteRow(room)
```

5.4 DirDBM: Directory-based Storage

5.4.1 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()
[]
```

5.4.2 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]
```

5.5 Using telnet to manipulate a twisted server

To start things off, we're going to create a simple server that just gives you remote access to a Python interpreter. We will use a telnet client to access this server.

Run `mktap telnet -p 4040 -u admin -w admin` at your shell prompt. If you list the contents of your current directory, you'll notice a new file – `telnet.tap`. After you do this, run `twistd -f telnet.tap`. Since the Application has a telnet server that you specified to be on port 4040, it will start listening for connections on this port. Try connecting with your favorite telnet utility to 127.0.0.1 port 4040.

```
$ telnet localhost 4040
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

twisted.manhole.telnet.ShellFactory
Twisted 1.1.0
username: admin
password: admin
>>>
```

Now, you should see a Python prompt – `>>>`. You can type any valid Python code here. Let's try looking around.

```
>>> dir()
['__builtins__']
```

Ok, not much. let's play a little more:

```
>>> import __main__
>>> dir(__main__)
['__builtins__', '__doc__', '__name__', 'os', 'run', 'string', 'sys']

>>> service
<twisted.application.internet.TCPServer instance at 0x10270f48>
>>> service._port
<twisted.manhole.telnet.ShellFactory on 4040>
>>> service.parent
<twisted.application.service.MultiService instance at 0x1024d7a8>
```

The service object is the service used to serve the telnet shell, and that it is listening on port 4040 with something called a `ShellFactory`. Its parent is a `twisted.application.service.MultiService`, a collection of services. We can keep getting the parent attribute of services until we hit the root of all services in this tap.

As you can see, this is quite useful - we can introspect a running process, see the internal objects, and even change their attributes. We can add telnet support to existing tap like so: `mktap --append=foo.tap telnet -p 4040 -u user -w pass`. The telnet server can of coursed be used from straight Python code as well. You can see how to do this by reading the code for `twisted.tap.telnet`.

A final note - if you want access to be more secure, you can even have the telnet server use SSL. Assuming you have the appropriate certificate and private key files, you can `mktap telnet -p ssl:443:privateKey=mykey.pem:certKey=cert.pem -u admin -w admin`. See `twisted.application.strports` for more examples of options for listening on a port.

Chapter 6

Perspective Broker

6.1 Introduction to Perspective Broker

6.1.1 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[1].

The Perspective Broker system (abbreviated "PB", spawning numerous sandwich-related puns) is based upon a few central concepts:

- *serialization*: taking fairly arbitrary objects and types, turning them into a chunk of bytes, sending them over a wire, then reconstituting them on the other end. By keeping careful track of object ids, the serialized objects can contain references to other objects and the remote copy will still be useful.
- *remote method calls*: doing something to a local object and causing a method to get run on a distant one. The local object is called a `RemoteReference`, and you "do something" by running its `.callRemote` method.

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.

6.1.2 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`

[1] Most of Twisted is like this. Hell, most of unix is like this: if *you* think it would be useful, someone else has probably thought that way in the past, and acted on it, and you can take advantage of the tool they created to solve the same problem you're facing now.

- *Broker*: `spread/pb.py`

Other classes that are involved at some point:

- *RemoteReference*: `spread/pb.py`
- *pb.Root*: `spread/pb.py`, actually defined as `Root` in `spread/flavors.py`
- *pb.Referenceable*: `spread/pb.py`, actually defined as `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.

XXX: add lists of useful-to-override methods here

6.1.3 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.flavors.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.flavors.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.

6.1.4 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.flavors.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.setCopierForClass(str(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.flavors.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.

6.2 Using Perspective Broker

6.2.1 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 `PBServerFactory` is given the root object, and the client retrieves it.

The client side uses `pb.PBClientFactory` to make a connection to a given port. This is a two-step process involving opening a TCP connection to a given host and port and requesting the root object using `.getRootObject()`.

Because `.getRootObject()` has to wait until a network connection has been made and exchange some data, it may take a while, so it returns a Deferred, to which the gotObject() callback is attached. (See the documentation on *Deferring Execution* (page 120) for a complete explanation of `Deferreds`). If and when the connection succeeds and a reference to the remote root object is obtained, this callback is run. The first argument passed to the callback is a remote reference to the distant root object. (you can give other arguments to the callback too, see the other parameters for `.addCallback()` and `.addCallbacks()`).

The callback does:

```
object.callRemote("echo", "hello network")
```

which causes the server's `.remote_echo()` method to be invoked. (running `.callRemote("boom")` would cause `.remote_boom()` to be run, etc). Again because of the delay involved, `callRemote()` returns a `Deferred`. Assuming the remote method was run without causing an exception (including an attempt to invoke an unknown method), the callback attached to that `Deferred` will be invoked with any objects that were returned by the remote method call.

In this example, the server's `Echoer` object has a method invoked, *exactly* as if some code on the server side had done:

```
echoer_object.remote_echo("hello network")
```

and from the definition of `remote_echo()` we see that this just returns the same string it was given: "hello network".

From the client's point of view, the remote call gets another `Deferred` object instead of that string. `callRemote()` *always* returns a `Deferred`. This is why PB is described as a system for "translucent" remote method calls instead of "transparent" ones: you cannot pretend that the remote object is really local. Trying to do so (as some other RPC mechanisms do, coughCORBAcough) breaks down when faced with the asynchronous nature of the network. Using Deferreds turns out to be a very clean way to deal with the whole thing.

The remote reference object (the one given to `getRootObject()`'s success callback) is an instance the `RemoteReference` class. This means you can use it to invoke methods on the remote object that it refers to. Only instances of `RemoteReference` are eligible for `.callRemote()`. The `RemoteReference` object is the one that lives on the remote side (the client, in this case), not the local side (where the actual object is defined).

In our example, the local object is that `Echoer()` instance, which inherits from `pb.Root`, which inherits from `pb.Referenceable`. It is that `Referenceable` class that makes the object eligible to be available for remote method calls[2]. 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`.)

[2] 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.

6.2.2 Complete Example

```
from twisted.spread import pb

class QuoteReader(pb.Root):

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

    def remote_nextQuote(self):
        return self.quoter.getQuote()
```

QuoteReader Root object — *pbquote.py*

For examples of these, we're returning to the TwistedQuotes project discussed in *Writing Plugins* (page 157). To use the examples in this HOWTO, we need to make a TML file to refer to our new set of examples:

```
register("Quote of the Day TAP Builder",
         "TwistedQuotes.quotetap2",
         description="""
         Example of a TAP builder module.
         """,
         type="tap",
         tapname="qotd")
```

Twisted Quotes Plug-in registration — *plugins2.tml*

The root object for TwistedQuotes is pretty small. The only thing it needs to keep track of for itself is the quoter object.

The QuoteReader publishes one method. By subclassing `Root`, we are declaring that all methods with the `remote_` prefix are remotely accessible.

In order to get this Root published, so that we can actually connect to it, we need to re-visit the TAP building plugin, so we can actually get an Application that has a PBServerFactory listening on a port. (The default port for PB is 8787.)

```
from TwistedQuotes import quoteproto    # Protocol and Factory
from TwistedQuotes import quoters       # "give me a quote" code
from TwistedQuotes import pbquote       # perspective broker binding

from twisted.application import service, internet
from twisted.python import usage        # twisted command-line processing
from twisted.spread import pb           # Perspective Broker

class Options(usage.Options):
    optParameters = [["port", "p", 8007,
                      "Port number to listen on for QOTD protocol."],
```

```
                    ["static", "s", "An apple a day keeps the doctor away.",
                     "A static quote to display."],
                    ["file", "f", None,
                     "A fortune-format text file to read quotes from."],
                    ["pb", "b", None,
                     "Port to listen with PB server"]]

def makeService(config):
    svc = service.MultiService()
    if config["file"]:                  # If I was given a "file" option...
        # Read quotes from a file, selecting a random one each time,
        quoter = quoters.FortuneQuoter([config['file']])
    else:                               # otherwise,
        # read a single quote from the command line (or use the default).
        quoter = quoters.StaticQuoter(config['static'])
    port = int(config["port"])          # TCP port to listen on
    factory = quoteproto.QOTDFactory(quoter) # here we create a QOTDFactory
    # Finally, set up our factory, with its custom quoter, to create QOTD
    # protocol instances when events arrive on the specified port.
    pbport = config['pb']               # TCP PB port to listen on
    if pbport:
        pbfact = pb.PBServerFactory(pbquote.QuoteReader(quoter))
        svc.addService(internet.TCPServer(int(pbport), pbfact))
    svc.addService(internet.TCPServer(port, factory))
    return svc
```

TAP Plugin with PB Quotes support — *quotetap2.py*

In the TAP builder, all we need to do is create our QuoteReader instance (making sure to pass it our quoter object), give it to a PBServerFactory, and create a TCPServer so that it can listen on a TCP port.

Accessing this through a client is fairly easy, as we use the `pb.PBClientFactory.getRootObject` method.

```
from sys import stdout
from twisted.python import log
log.discardLogs()
from twisted.internet import reactor
from twisted.spread import pb

def connected(root):
    root.callRemote('nextQuote').addCallbacks(success, failure)

def success(quote):
    stdout.write(quote + "\n")
    reactor.stop()
```

```
def failure(error):
    stdout.write("Failed to obtain quote.\n")
    reactor.stop()

factory = pb.PBClientFactory()
reactor.connectTCP(
    "localhost", # host name
    pb.portno, # port number
    factory, # factory
    )



factory.getRootObject().addCallbacks(connected, # when we get the root
                                     failure)   # when we can't

reactor.run() # start the main loop
```

PB Quotes Client Code — *pbquoteclient.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.

In this example, the `connected` callback should be made when the script is run. Looking at the code, it should be clear that in the event of a connection success, the client will print out a quote and exit. If you start up a server, you can see:

```
% mktap qotd --pb 8787
% twistd -f qotd.tap
% python -c 'import TwistedQuotes.pbquoteclient'
An apple a day keeps the doctor away.
```

The argument to this callback, `root`, is a `RemoteReference`. It represents a reference to the `QuoteReader` object.

`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.

This introduction to PB does not showcase all of the features that it provides, but hopefully it gives you a good idea of where to get started setting up your own application. Here are some of the other building blocks you can use.

6.2.3 Passing more references

Here is an example of using `pb.Referenceable` in a second class. The second `Referenceable` object can have remote methods invoked too, just like the first. In this example, the initial root object has a method that returns a reference to the second object.

```
#! /usr/bin/python
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()))
```

Source listing — *pb1server.py*

```
#! /usr/bin/python

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

def main():
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 8800, factory)
    def1 = factory.getRootObject()
    def1.addCallbacks(got_obj1, err_obj1)
    reactor.run()

def err_obj1(reason):
    print "error getting first object", reason
    reactor.stop()

def got_obj1(obj1):
    print "got first object:", obj1
    print "asking it to getTwo"
    def2 = obj1.callRemote("getTwo")
    def2.addCallbacks(got_obj2)

def got_obj2(obj2):
    print "got second object:", obj2
    print "telling it to do three(12)"
```

```
    obj2.callRemote("three", 12)

main()
```

Source listing — *pb1client.py*

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.

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[3].

6.2.4 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()`[4].

```
#! /usr/bin/python

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

class Two(pb.Referenceable):
    def remote_print(self, arg):
        print "two.print was given", arg

class One(pb.Root):
    def __init__(self, two):
        #pb.Root.__init__(self)   # pb.Root doesn't implement __init__
        self.two = two
    def remote_getTwo(self):
        print "One.getTwo(), returning my two called", two
        return two
    def remote_checkTwo(self, newtwo):
        print "One.checkTwo(): comparing my two", self.two
        print "One.checkTwo(): against your two", newtwo
        if two == newtwo:
            print "One.checkTwo(): our twos are the same"
```

[3]This can be overridden, by subclassing one of the Serializable flavors and defining custom serialization code for your class. See *Passing Complex Types* (page 213) for details.

[4]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.

```
two = Two()
root_obj = One(two)
reactor.listenTCP(8800, pb.PBServerFactory(root_obj))
reactor.run()
```

Source listing — *pb2server.py*

```
#! /usr/bin/python

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 120) 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.

6.2.5 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/python

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/python

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.

6.2.6 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[5]. 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/python
```

[5] 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.

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

class MyError(pb.Error):
    """This is an Expected Exception. Something bad happened."""
    pass

class MyError2(Exception):
    """This is an Unexpected Exception. Something really bad happened."""
    pass

class One(pb.Root):
    def remote_broken(self):
        msg = "fall down go boom"
        print "raising a MyError exception with data '%s'" % msg
        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/python

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`.

6.2.7 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/python

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/python

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 `pb.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[6]. The receiving end of the connection gets to decide what to accept and what to reject. It indicates its disapproval by raising a `pb.InsecureJelly` exception. Because it occurs at the remote end, the exception is

[6]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.

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.

6.3 PB Copyable: Passing Complex Types

6.3.1 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).

6.3.2 Motivation

From the *previous chapter* (page 197), 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.

The pb.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 213) for an example.

6.3.3 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 specific 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[7]. 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.

6.3.4 pb.Copyable

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

```
#! /usr/bin/python

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
```

[7] 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".

```
class Sender:
    def __init__(self, pond):
        self.pond = pond

    def got_obj(self, remote):
        self.remote = remote
        d = remote.callRemote("takePond", self.pond)
        d.addCallback(self.ok).addErrback(self.notOk)

    def ok(self, response):
        print "pond arrived", response
        reactor.stop()
    def notOk(self, failure):
        print "error during takePond:"
        if failure.type == jelly.InsecureJelly:
            print " InsecureJelly"
        else:
            print failure
        reactor.stop()
        return None

def main():
    from copy_sender import CopyPond  # so it's not __main__.CopyPond
    pond = CopyPond()
    pond.setStuff("green", 7)
    pond.countFrogs()
    # class name:
    print ".".join([pond.__class__.__module__, pond.__class__.__name__])

    sender = Sender(pond)
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 8800, factory)
    deferred = factory.getRootObject()
    deferred.addCallback(sender.got_obj)
    reactor.run()

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

Source listing — *copy_sender.py*

```
#! /usr/bin/python

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
import sys
#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.py*

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 227)).

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/python

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/python

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/python

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. (XXX: this works, but does this really get the right names into the table? Or does it only work because both are defined in the same (wrong) place?)
- The class that is sent must inherit from pb.Copyable. The class that is registered to receive it must inherit from pb.RemoteCopy[8].
- 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. XXX: check this, the code around jelly._Unjellier.unjelly:489 tries to avoid calling __init__ just in case the constructor requires args.

[8] pb.RemoteCopy is actually defined as flavors.RemoteCopy, but pb.RemoteCopy is the preferred way to access it

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

6.3.5 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[9].

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"[10] 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*

[9] 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 result might be hard to maintain or extend, though.

[10] 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.

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/python

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/python

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/python

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[11].
- 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 `callRemotes` to throw away such an error. This is a useful idiom:

  ```
  observer.callRemote('foo', arg).addErrback(lambda f: None)
  ```

  (XXX: verify that this is actually a concern)
- `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.

[11] 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.

XXX: understand, then explain use of varying cached state depending upon perspective.

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. (XXX: I've heard explorer is currently broken, it might not be a good example to recommend)
- The `spread.publish` module also uses `Cacheable`, and might be a source of further information.

6.4 Authentication with Perspective Broker

6.4.1 Overview

The examples shown in *Using Perspective Broker* (page 197) 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 149) to handle authentication issues, and Perspective Broker has code to make it easy to implement the most common use cases.

6.4.2 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.RemoteReferences` 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 197). 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.[12]

(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[13], 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):
```

[12] 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.

[13] 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*

```
        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:**
> 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 were 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)
    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 message.find("mattress") != -1:
            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 not self.groups.has_key(groupname):
            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? Also look for "banana" and "marmalade". 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 message.find("mattress") != -1:
            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 240).

6.4.3 Avatars and Perspectives

In Twisted's *cred* (page 149) 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[14]. Just remember that perspectives and avatars are basically the same thing.

Despite all we've been *telling you* (page 149) 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`[15]. 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 149), 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.

---

[14]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.

[15]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.

6.4.4 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/python

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())
p.registerChecker(checkers.InMemoryDatabaseDontUse(user1="pass1"))
reactor.listenTCP(8800, pb.PBServerFactory(p))
reactor.run()
```

Source listing — *pb5server.py*

```
#! /usr/bin/python

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/python

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/python

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/python

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 `InMemoryDatabaseDontUse`, 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 197)) 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.

6.4.5 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/python

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 message.find("mattress") != -1:
            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/python

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 #lookingForFourth"
        # this perspective is a reference to our User object
        d = perspective.callRemote("joinGroup", "#lookingForFourth")
        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*

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

6.5 Managing Clients of Perspectives

6.5.1 Overview

In all the `IPerspective` 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.

6.5.2 Managing Avatars

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

```
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:

```
class Greeter:
    greetings = 0
    def hello(self):
        self.greetings += 1
        return "<%d>hello" % (self.greetings, self.name)

class SimpleAvatar(pb.Avatar):
    greetings = 0
    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:

```
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:

```
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
```

6.5.3 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:

```
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.

Chapter 7

Web Applications

7.1 Configuring and Using the Twisted.Web Server

7.1.1 Twisted Web Development

Twisted Web serves python objects that implement the interface IResource.

Main Concepts

- *Site Objects* (this page) are responsible for creating `HTTPChannel` instances to parse the HTTP request, and begin the object lookup process. They contain the root Resource, the resource which represents the URL `/` on the site.
- *Resource* (page 248) objects represent a single URL segment. The `IResource` interface describes the methods a Resource object must implement in order to participate in the object publishing process.
- *Resource trees* (page 249) are arrangements of Resource objects into a Resource tree. Starting at the root Resource object, the tree of Resource objects defines the URLs which will be valid.
- *.rpy scripts* (page 249) are python scripts which the twisted.web static file server will execute, much like a CGI. However, unlike CGI they must create a Resource object which will be rendered when the URL is visited.
- *Resource rendering* (page 250) occurs when Twisted Web locates a leaf Resource object. A Resource can either return an html string or write to the request object.
- *Session* (page 250) objects allow you to store information across multiple requests. Each individual browser using the system has a unique Session instance.

Site Objects

Site objects serve as the glue between a port to listen for HTTP requests on, and a root Resource object.

When using `mktap web --path /foo/bar/baz`, a Site object is created with a root Resource that serves files out of the given path.

You can also create a `Site` instance by hand, passing it a `Resource` object which will serve as the root of the site:

```
from twisted.web import server, resource
from twisted.internet import reactor

class Simple(resource.Resource):
    isLeaf = True
    def render_GET(self, request):
        return "<html>Hello, world!</html>"

site = server.Site(Simple())
reactor.listenTCP(8080, site)
reactor.run()
```

Resource objects

`Resource` objects represent a single URL segment of a site. During URL parsing, `getChild` is called on the current `Resource` to produce the next `Resource` object.

When the leaf Resource is reached, either because there were no more URL segments or a Resource had isLeaf set to True, the leaf Resource is rendered by calling `render(request)`. See “Resource Rendering” below for more about this.

During the Resource location process, the URL segments which have already been processed and those which have not yet been processed are available in `request.prepath` and `request.postpath`.

A Resource can know where it is in the URL tree by looking at `request.prepath`, a list of URL segment strings.

A Resource can know which path segments will be processed after it by looking at `request.postpath`.

If the URL ends in a slash, for example `http://example.com/foo/bar/`, the final URL segment will be an empty string. Resources can thus know if they were requested with or without a final slash.

Here is a simple Resource object:

```
from twisted.web.resource import Resource

class Hello(Resource):
    def getChild(self, name, request):
        if name == '':
            return self
        return Resource.getChild(
            self, name, request)

        def render_GET(self, request):
            return """<html>
      Hello, world! I am located at %r.
    </html>""" % (request.prepath)

resource = Hello()
```

Resource Trees

Resources can be arranged in trees using `putChild`. `putChild` puts a Resource instance into another Resource instance, making it available at the given path segment name:

```
root = Hello()
root.putChild('fred', Hello())
root.putChild('bob', Hello())
```

If this root resource is served as the root of a Site instance, the following URLs will all be valid:

- `http://example.com/`
- `http://example.com/fred`
- `http://example.com/bob`
- `http://example.com/fred/`
- `http://example.com/bob/`

.rpy scripts

Files with the extension `.rpy` are python scripts which, when placed in a directory served by Twisted Web, will be executed when visited through the web.

An `.rpy` script must define a variable, `resource`, which is the Resource object that will render the request.

`.rpy` files are very convenient for rapid development and prototyping. Since they are executed on every web request, defining a Resource subclass in an `.rpy` will make viewing the results of changes to your class visible simply by refreshing the page:

```
from twisted.web.resource import Resource

class MyResource(Resource):
    def render_GET(self, request):
        return "<html>Hello, world!</html>"

resource = MyResource()
```

However, it is often a better idea to define Resource subclasses in Python modules. In order for changes in modules to be visible, you must either restart the Python process, or reload the module:

```
import myresource

## Comment out this line when finished debugging
reload(myresource)

resource = myresource.MyResource()
```

Creating a Twisted Web server which serves a directory is easy:

```
% mktap web --path /Users/dsp/Sites
% twistd -nf web.tap
```

Resource rendering

Resource rendering occurs when Twisted Web locates a leaf Resource object to handle a web request. A Resource's `render` method may do various things to produce output which will be sent back to the browser:

- Return a string
- Call `request.write("stuff")` as many times as desired, then call `request.finish()` and return `server.NOT_DONE_YET` (This is deceptive, since you are in fact done with the request, but is the correct way to do this)
- Request a `Deferred`, return `server.NOT_DONE_YET`, and call `request.write("stuff")` and `request.finish()` later, in a callback on the `Deferred`.

The `Resource` class, which is usually what one's Resource classes subclass, has a convenient default implementation of `render`. It will call a method named `self.render_METHOD` where "METHOD" is whatever HTTP method was used to request this resource. Examples: request_GET, request_POST, request_HEAD, and so on. It is recommended that you have your resource classes subclass `Resource` and implement `render_METHOD` methods as opposed to `render` itself. Note that for certain resources, `request_POST = request_GET` may be desirable in case one wants to process arguments passed to the resource regardless of whether they used GET (`?foo=bar&baz=quux`, and so forth) or POST.

Session

HTTP is a stateless protocol; every request-response is treated as an individual unit, distinguishable from any other request only by the URL requested. With the advent of Cookies in the mid nineties, dynamic web servers gained the ability to distinguish between requests coming from different *browser sessions* by sending a Cookie to a browser. The browser then sends this cookie whenever it makes a request to a web server, allowing the server to track which requests come from which browser session.

Twisted Web provides an abstraction of this browser-tracking behavior called the *Session object*. Calling `request.getSession()` checks to see if a session cookie has been set; if not, it creates a unique session id, creates a Session object, stores it in the Site, and returns it. If a session object already exists, the same session object is returned. In this way, you can store data specific to the session in the session object.

7.1.2 Advanced Configuration

Non-trivial configurations of Twisted Web are achieved with Python configuration files. This is a Python snippet which builds up a variable called application. Usually, a `twisted.application.internet.TCPServer` instance will be used to make the application listen on a TCP port (80, in case direct web serving is desired), with the listener being a `twisted.web.server.Site`. The resulting file can then be run with `twistd -y`. Alternatively a reactor object can be used directly to make a runnable script.

The `Site` will wrap a `Resource` object – the root.

```
from twisted.application import internet, service
from twisted.web import static, server

root = static.File("/var/www/htdocs")
application = service.Application('web')
```

```
site = server.Site(root)
sc = service.IServiceCollection(application)
i = internet.TCPServer(80, site)
i.setServiceParent(sc)
```

Most advanced configurations will be in the form of tweaking the root resource object.

Adding Children

Usually, the root's children will be based on the filesystem's contents. It is possible to override the filesystem by explicit `putChild` methods.

Here are two examples. The first one adds a `/doc` child to serve the documentation of the installed packages, while the second one adds a `cgi-bin` directory for CGI scripts.

```
from twisted.internet import reactor
from twisted.web import static, server

root = static.File("/var/www/htdocs")
root.putChild("doc", static.File("/usr/share/doc"))
reactor.listenTCP(80, server.Site(root))
reactor.run()
```

```
from twisted.internet import reactor
from twisted.web import static, server, twcgi

root = static.File("/var/www/htdocs")
root.putChild("cgi-bin", twcgi.CGIDirectory("/var/www/cgi-bin"))
reactor.listenTCP(80, server.Site(root))
reactor.run()
```

Modifying File Resources

`File` resources, be they root object or children thereof, have two important attributes that often need to be modified: `indexNames` and `processors`. `indexNames` determines which files are treated as "index files" – served up when a directory is rendered. `processors` determine how certain file extensions are treated.

Here is an example for both, creating a site where all `.rpy` extensions are Resource Scripts, and which renders directories by searching for a `index.rpy` file.

```
from twisted.application import internet, service
from twisted.web import static, server, script

root = static.File("/var/www/htdocs")
root.indexNames=['index.rpy']
root.processors = {'.rpy': script.ResourceScript}
application = service.Application('web')
sc = service.IServiceCollection(application)
site = server.Site(root)
```

```
i = internet.TCPServer(80, site)
i.setServiceParent(sc)
```

`File` objects also have a method called `ignoreExt`. This method can be used to give extension-less URLs to users, so that implementation is hidden. Here is an example:

```
from twisted.application import internet, service
from twisted.web import static, server, script

root = static.File("/var/www/htdocs")
root.ignoreExt(".rpy")
root.processors = {'.rpy': script.ResourceScript}
application = service.Application('web')
sc = service.IServiceCollection(application)
site = server.Site(root)
i = internet.TCPServer(80, site)
i.setServiceParent(sc)
```

Now, a URL such as `/foo` might be served from a Resource Script called `foo.rpy`, if no file by the name of `foo` exists.

Virtual Hosts

Virtual hosting is done via a special resource, that should be used as the root resource – `NameVirtualHost`. `NameVirtualHost` has an attribute named `default`, which holds the default website. If a different root for some other name is desired, the `addHost` method should be called.

```
from twisted.application import internet, service
from twisted.web import static, server, vhost, script

root = vhost.NameVirtualHost()

# Add a default -- htdocs
root.default=static.File("/var/www/htdocs")

# Add a simple virtual host -- foo.com
root.addHost("foo.com", static.File("/var/www/foo"))

# Add a simple virtual host -- bar.com
root.addHost("bar.com", static.File("/var/www/bar"))

# The "baz" people want to use Resource Scripts in their web site
baz = static.File("/var/www/baz")
baz.processors = {'.rpy': script.ResourceScript}
baz.ignoreExt('.rpy')
root.addHost('baz', baz)

application = service.Application('web')
```

```
sc = service.IServiceCollection(application)
site = server.Site(root)
i = internet.TCPServer(80, site)
i.setServiceParent(sc)
```

Advanced Techniques

Since the configuration is a Python snippet, it is possible to use the full power of Python. Here are some simple examples:

```
# No need for configuration of virtual hosts -- just make sure
# a directory /var/vhosts/<vhost name> exists:
from twisted.web import vhost, static, server
from twisted.application import internet, service

root = vhost.NameVirtualHost()
root.default = static.File("/var/www/htdocs")
for dir in os.listdir("/var/vhosts"):
    root.addHost(dir, static.File(os.path.join("/var/vhosts", dir)))

application = service.Application('web')
sc = service.IServiceCollection(application)
site = server.Site(root)
i = internet.TCPServer(80, site)
i.setServiceParent(sc)

# Determine ports we listen on based on a file with numbers:
from twisted.web import vhost, static, server
from twisted.application import internet, service

root = static.File("/var/www/htdocs")

site = server.Site(root)
application = service.Application('web')
serviceCollection = service.IServiceCollection(application)

for num in map(int, open("/etc/web/ports").read().split()):
    serviceCollection.addCollection(internet.TCPServer(num, site))
```

7.1.3 Installing a pre-configured server

In many cases, you'll end up repeating common usage patterns of twisted.web. In those cases you'll probably want to use Twisted's pre-configured web server setup.

To install the Twisted.Web server, you'll need to have installed Twisted.

Pre-configured Twisted servers, like the web server, do not have configuration files. Instead, you instantiate the server and store it into a 'Pickle' file, `web.tap`. This file will then be loaded by the Twisted Daemon.

```
% mktap web --path /path/to/web/content
```

If you just want to serve content from your own home directory, the following will do:

```
% mktap web --path ~/public_html/
```

Some other configuration options are available as well:

- `--port`: Specify the port for the web server to listen on. This defaults to 8080.
- `--logfile`: Specify the path to the log file.

The full set of options that are available can be seen with:

```
% mktap web --help
```

7.1.4 Using Twisted.Web

Stopping and Starting the Server

Once you've created your `web.tap` file and done any configuration, you can start the server:

```
% twistd -f web.tap
```

You can stop the server at any time by going back to the directory you started it in and running the command:

```
% kill `cat twistd.pid`
```

Serving Flat HTML

Twisted.Web serves flat HTML files just as it does any other flat file.

Resource Scripts

A Resource script is a Python file ending with the extension `.rpy`, which is required to create an instance of a (subclass of a) `twisted.web.resource.Resource`.

Resource scripts have 3 special variables:

- `__file__`: The name of the .rpy file, including the full path. This variable is automatically defined and present within the namespace.
- `registry`: An object of class `static.Registry`. It can be used to access and set persistent data keyed by a class.
- `resource`: The variable which must be defined by the script and set to the resource instance that will be used to render the page.

A very simple Resource Script might look like:

```
from twisted.web import resource
class MyGreatResource(resource.Resource):
    def render_GET(self, request):
        return "<html>foo</html>"

resource = MyGreatResource()
```

A slightly more complicated resource script, which accesses some persistent data, might look like:

```
from twisted.web import resource
from SillyWeb import Counter

counter = registry.getComponent(Counter)
if not counter:
   registry.setComponent(Counter, Counter())
counter = registry.getComponent(Counter)

class MyResource(resource.Resource):
    def render_GET(self, request):
        counter.increment()
        return "you are visitor %d" % counter.getValue()

resource = MyResource()
```

This is assuming you have the `SillyWeb.Counter` module, implemented something like the following:

```
class Counter:

    def __init__(self):
        self.value = 0

    def increment(self):
        self.value += 1

    def getValue(self):
        return self.value
```

Web UIs

The Nevow[1] framework, available as part of the Quotient[2] project, is an advanced system for giving Web UIs to your application. Nevow uses Twisted Web but is not itself part of Twisted.

Spreadable Web Servers

One of the most interesting applications of Twisted.Web is the distributed webserver; multiple servers can all answer requests on the same port, using the `twisted.spread` package for "spreadable" computing. In two different directories, run the commands:

[1] http://www.divmod.org/Home/Projects/Nevow/
[2] http://www.divmod.org/Quotient/

```
% mktap web --user
% mktap web --personal [other options, if you desire]
```

Both of these create a `web.tap`; you need to run both at the same time. Once you have, go to `http://localhost:8080/your_username.twistd/` – you will see the front page from the server you created with the `--personal` option. What's happening here is that the request you've sent is being relayed from the central (User) server to your own (Personal) server, over a PB connection. This technique can be highly useful for small "community" sites; using the code that makes this demo work, you can connect one HTTP port to multiple resources running with different permissions on the same machine, on different local machines, or even over the internet to a remote site.

Serving PHP/Perl/CGI

Everything related to CGI is located in the `twisted.web.twcgi`, and it's here you'll find the classes that you need to subclass in order to support the language of your (or somebody elses) taste. You'll also need to create your own kind of resource if you are using a non-unix operatingsystem (such as Windows), or if the default resources has wrong pathnames to the parsers.

The following snippet is a .rpy that serves perl-files. Look at `twisted.web.twcgi` for more examples regarding twisted.web and CGI.

```
from twisted.web import static, twcgi

class PerlScript(twcgi.FilteredScript):
    filter = '/usr/bin/perl' # Points to the perl parser

resource = static.File("/perlsite") # Points to the perl website
resource.processors = {".pl": PerlScript} # Files that end with .pl will be
                                          # processed by PerlScript
resource.indexNames = ['index.pl']
```

Using VHostMonster

It is common to use one server (for example, Apache) on a site with multiple names which then uses reverse proxy (in Apache, via `mod_proxy`) to different internal web servers, possibly on different machines. However, naive configuration causes miscommunication: the internal server firmly believes it is running on "internal-name:port", and will generate URLs to that effect, which will be completely wrong when received by the client.

While Apache has the ProxyPassReverse directive, it is really a hack and is nowhere near comprehensive enough. Instead, the recommended practice in case the internal web server is Twisted.Web is to use VHostMonster.

From the Twisted side, using VHostMonster is easy: just drop a file named (for example) `vhost.rpy` containing the following:

```
from twisted.web import vhost
resource = vhost.VHostMonsterResource()
```

Of course, an equivalent `.trp` can also be used. Make sure the web server is configured with the correct processors for the `rpy` or `trp` extensions (the web server `mktap web --path` generates by default is so configured).

From the Apache side, instead of using the following ProxyPass directive:

```
<VirtualHost ip-addr>
ProxyPass / http://localhost:8538/
ServerName example.com
</VirtualHost>
```

Use the following directive:

```
<VirtualHost ip-addr>
ProxyPass / http://localhost:8538/vhost.rpy/http/example.com:80/
ServerName example.com
</VirtualHost>
```

Here is an example for Twisted.Web's reverse proxy:

```
from twisted.application import internet, service
from twisted.web import proxy, server, vhost
vhostName = 'example.com'
reverseProxy = proxy.ReverseProxyResource('internal', 8538,
                                          '/vhost.rpy/http/'+vhostName+'/')
root = vhost.NameVirtualHost()
root.addHost(vhostName, reverseProxy)
site = server.Site(root)
application = service.Application('web-proxy')
sc = service.IServiceCollection(application)
i = internet.TCPServer(80, site)
i.setServiceParent(sc)
```

7.1.5 Rewriting URLs

Sometimes it is convenient to modify the content of the `Request` object before passing it on. Because this is most often used to rewrite either the URL, the similarity to Apache's `mod_rewrite` has inspired the `twisted.web.rewrite` module. Using this module is done via wrapping a resource with a `twisted.web.rewrite.RewriterResource` which then has rewrite rules. Rewrite rules are functions which accept a request object, and possible modify it. After all rewrite rules run, the child resolution chain continues as if the wrapped resource, rather than the `RewriterResource`, was the child.

Here is an example, using the only rule currently supplied by Twisted itself:

```
default_root = rewrite.RewriterResource(default, rewrite.tildeToUsers)
```

This causes the URL `/~foo/bar.html` to be treated like `/users/foo/bar.html`. If done after setting default's `users` child to a `distrib.UserDirectory`, it gives a configuration similar to the classical configuration of web server, common since the first NCSA servers.

7.1.6 Knowing When We're Not Wanted

Sometimes it is useful to know when the other side has broken the connection. Here is an example which does that:

```
from twisted.web.resource import Resource
from twisted.web import server
from twisted.internet import reactor
from twisted.python.util import println


class ExampleResource(Resource):

    def render_GET(self, request):
        request.write("hello world")
        d = request.notifyFinish()
        d.addCallback(lambda _: println("finished normally"))
        d.addErrback(println, "error")
        reactor.callLater(10, request.finish)
        return server.NOT_DONE_YET

resource = ExampleResource()
```

This will allow us to run statistics on the log-file to see how many users are frustrated after merely 10 seconds.

7.1.7 As-Is Serving

Sometimes, you want to be able to send headers and status directly. While you can do this with a `ResourceScript`, an easier way is to use `AsIsProcessor`. Use it by, for example, addding it as a processor for the `.asis` extension. Here is a sample file:

```
HTTP/1.0 200 OK
Content-Type: text/html

Hello world
```

7.2 Web Application Development

7.2.1 Code layout

The development of a Twisted Web application should be orthogonal to its deployment. This means is that if you are developing a web application, it should be a resource with children, and internal links. Some of the children might use Nevow[3], some might be resources manually using `.write`, and so on. Regardless, the code should be in a Python module, or package, *outside* the web tree.

You will probably want to test your application as you develop it. There are many ways to test, including dropping an `.rpy` which looks like:

```
from mypackage import toplevel
resource = toplevel.Resource(file="foo/bar", color="blue")
```

into a directory, and then running:

[3] http://www.divmod.org/Home/Projects/Nevow/

```
% mktap web --path=/directory
% twistd -f web.tap
```

You can also write a Python script like:

```
#!/usr/bin/python2.2

from twisted.web import server
from twisted.internet import reactor
from mypackage import toplevel

reactor.listenTCP(8080,
    server.Site(toplevel.Resource(file="foo/bar", color="blue")))
reactor.run()
```

7.2.2 Web application deployment

Which one of these development strategies you use is not terribly important, since (and this is the important part) deployment is *orthogonal*. Later, when you want users to actually *use* your code, you should worry about what to do – or rather, don't. Users may have widely different needs. Some may want to run your code in a different process, so they'll use distributed web (`twisted.web.distrib`). Some may be using the `twisted-web` Debian package, and will drop in:

```
% cat > /etc/local.d/99addmypackage.py
from mypackage import toplevel
default.putChild("mypackage", toplevel.Resource(file="foo/bar", color="blue"))
^D
```

If you want to be friendly to your users, you can supply many examples in your package, like the above `.rpy` and the Debian-package drop-in. But the *ultimate* friendliness is to write a useful resource which does not have deployment assumptions built in.

7.2.3 Understanding resource scripts (.rpy files)

Twisted Web is not PHP – it has better tools for organizing code Python modules and packages, so use them. In PHP, the only tool for organizing code is a web page, which leads to silly things like PHP pages full of functions that other pages import, and so on. If you were to write your code this way with Twisted Web, you would do web development using many `.rpy` files, all importing some Python module. This is a *bad idea* – it mashes deployment with development, and makes sure your users will be *tied* to the file-system.

We have `.rpy`s because they are useful and necessary. But using them incorrectly leads to horribly unmaintainable applications. The best way to ensure you are using them correctly is to not use them at all, until you are on your *final* deployment stages. You should then find your `.rpy` files will be less than 10 lines, because you will not *have* more than 10 lines to write.

7.3 Light Weight Templating With Resource Templates

7.3.1 Overview

While Twisted supports solution like the Nevow[4] framework (available as part of the Quotient[5] project) for high-content sophisticated templating needs, sometimes one needs a less file-heavy system which lets one directly write HTML. While ResourceScripts are available, they have a high overhead of coding, needing some boring string arithmetic. ResourceTemplates fill the space between *Woven* (page 267) and ResourceScript using Quixote's PTL (Python Templating Language).

ResourceTemplates need Quixote installed. In Debian[6], that means using Python 2.2 and installing the `quixote` package (`apt-get install quixote`). Other operating systems require other ways to install quixote, or it can be done manually.

7.3.2 Configuring Twisted.Web

The easiest way to get Twisted.Web to support ResourceTemplates is to bind them to some extension using the web tap's `--processor` flag. Here is an example:

```
% mktap web --path=/var/www \
        --processor=.rtl=twisted.web.script.ResourceTemplate
```

The above command line binds the `rtl` extension to use the ResourceTemplate processor. Other ways are possible, but would require more Python coding and are outside the scope of this HOWTO.

7.3.3 Using ResourceTemplate

ResourceTemplates are coded in an extension of Python called the "Python Templating Language". Complete documentation of the PTL is available at the quixote web site[7]. The web server will expect the PTL source file to define a variable named `resource`. This should be a `twisted.web.server.Resource`, whose `.render` method be called. Usually, you would want to define `render` using the keyword `template` rather than `def`.

Here is a simple example for a resource template.

```
from twisted.web.resource import Resource
from TwistedQuotes import quoters

quotefile = os.path.join(os.path.split(__file__)[0], "quotes.txt")

quoter = quoters.FortuneQuoter([quotefile])


class QuoteResource(Resource):

    template render(self, request):
```

4 http://www.divmod.org/Home/Projects/Nevow/
5 http://www.divmod.org/Quotient/
6 http://www.debian.org
7 http://www.mems-exchange.org/software/quixote/doc/PTL.html

```
        """\
        <html>
        <head><title>Quotes Galore</title></head>

        <body><h1>Quotes</h1>"""
        quoter.getQuote()
        "</body></html>"


resource = QuoteResource()
```

Resource Template for Quotes — *webquote.rtl*

7.4 Creating XML-RPC Servers and Clients with Twisted

7.4.1 Introduction

XML-RPC[8] is a simple request/reply protocol that runs over HTTP. It is simple, easy to implement and supported by most programming languages. Twisted's XML-RPC support is implemented using the xmlrpclib library that is included with Python 2.2 and later.

7.4.2 Creating a XML-RPC server

Making a server is very easy - all you need to do is inherit from `twisted.web.xmlrpc.XMLRPC`. You then create methods beginning with `xmlrpc_`. The methods' arguments determine what arguments it will accept from XML-RPC clients. The result is what will be returned to the clients.

Methods published via XML-RPC can return all the basic XML-RPC types, such as strings, lists and so on (just return a regular python integer, etc). They can also return Failure instances to indicate an error has occurred, or `Binary`, `Boolean` or `DateTime` instances (all of these are the same as the respective classes in xmlrpclib. In addition, XML-RPC published methods can return `Deferred` instances whose results are one of the above. This allows you to return results that can't be calculated immediately, such as database queries. See the *Deferred documentation* (page 120) for more details.

`XMLRPC` instances are Resource objects, and they can thus be published using a Site. The following example has two methods published via XML-RPC, `add(a, b)` and `echo(x)`.

```
from twisted.web import xmlrpc, server

class Example(xmlrpc.XMLRPC):
    """An example object to be published."""

    def xmlrpc_echo(self, x):
        """Return all passed args."""
```

[8] http://www.xmlrpc.com

```
        return x

    def xmlrpc_add(self, a, b):
        """Return sum of arguments."""
        return a + b

if __name__ == '__main__':
    from twisted.internet import reactor
    r = Example()
    reactor.listenTCP(7080, server.Site(r))
    reactor.run()
```

After we run this command, we can connect with a client and send commands to the server:

```
>>> import xmlrpclib
>>> s = xmlrpclib.Server('http://localhost:7080/')
>>> s.echo("lala")
'lala'
>>> s.add(1, 2)
3
```

XML-RPC resources can also be part of a normal Twisted web server, using resource scripts. The following is an example of such a resource script:

```
from twisted.web import xmlrpc
from TwistedQuotes import quoters
import os

quotefile = os.path.join(os.path.split(__file__)[0], "quotes.txt")
quoter = quoters.FortuneQuoter([quotefile])

class Quoter(xmlrpc.XMLRPC):

    def xmlrpc_quote(self):
        return quoter.getQuote()

resource = Quoter()
```

Source listing — *xmlquote.rpy*

Using XML-RPC sub-handlers

XML-RPC resource can be nested so that one handler calls another if a method with a given prefix is called. For example, to add support for an XML-RPC method `date.time()` to the `Example` class, you could do the following:

```
import time
from twisted.web import xmlrpc, server
```

```
class Example(xmlrpc.XMLRPC):
    """An example object to be published."""

    def xmlrpc_echo(self, x):
        """Return all passed args."""
        return x

    def xmlrpc_add(self, a, b):
        """Return sum of arguments."""
        return a + b

class Date(xmlrpc.XMLRPC):
    """Serve the XML-RPC 'time' method."""

    def xmlrpc_time(self):
        """Return UNIX time."""
        return time.time()

if __name__ == '__main__':
    from twisted.internet import reactor
    r = Example()
    date = Date()
    r.putSubHandler('date', date)
    reactor.listenTCP(7080, server.Site(r))
    reactor.run()
```

By default, a period ('.') separates the prefix from the method name, but you can use a different character by overriding the `XMLRPC.separator` data member in your base XML-RPC server. XML-RPC servers may be nested to arbitrary depths using this method.

Adding XML-RPC Introspection support

XML-RPC has an informal Introspection API[9] that specifies three methods in a `system` sub-handler which allow a client to query a server about the server's API. Adding Introspection support to the `Example` class is easy using the `XMLRPCIntrospection` class:

```
from twisted.web import xmlrpc, server

class Example(xmlrpc.XMLRPC):
    """An example object to be published."""

    def xmlrpc_echo(self, x):
        """Return all passed args."""
        return x
```

[9] http://ldp.kernelnotes.de/HOWTO/XML-RPC-HOWTO/xmlrpc-howto-interfaces.html

```
    xmlrpc_echo.signature = [['string', 'string'],
                             ['int', 'int'],
                             ['double', 'double'],
                             ['array', 'array'],
                             ['struct', 'struct']]

    def xmlrpc_add(self, a, b):
        """Return sum of arguments."""
        return a + b

    xmlrpc_add.signature = [['int', 'int', 'int'],
                            ['double', 'double', 'double']]
    xmlrpc_add.help = "Add the arguments and return the sum."

if __name__ == '__main__':
    from twisted.internet import reactor
    r = Example()
    xmlrpc.addIntrospection(r)
    reactor.listenTCP(7080, server.Site(r))
    reactor.run()
```

Note the method attributes `help` and `signature` which are used by the Introspection API methods `system.methodHelp` and `system.methodSignature` respectively. If no `help` attribute is specified, the method's documentation string is used instead.

7.4.3 SOAP Support

From the point of view of a Twisted developer, there is little difference between XML-RPC support and SOAP support. Here is an example of SOAP usage:

```
from twisted.web import soap
from TwistedQuotes import quoters
import os

quotefile = os.path.join(os.path.split(__file__)[0], "quotes.txt")
quoter = quoters.FortuneQuoter([quotefile])

class Quoter(soap.SOAPPublisher):
    """Publish one method, 'quote'."""

    def soap_quote(self):
        return quoter.getQuote()

resource = Quoter()
```

Source listing — *soap.rpy*

7.4.4 Creating an XML-RPC Client

XML-RPC clients in Twisted are meant to look as something which will be familiar either to `xmlrpclib` or to Perspective Broker users, taking features from both, as appropriate. There are two major deviations from the `xmlrpclib` way which should be noted:

1. No implicit `/RPC2`. If the services uses this path for the XML-RPC calls, then it will have to be given explicitly.
2. No magic `__getattr__`: calls must be made by an explicit `callRemote`.

The interface Twisted presents to XML-RPC client is that of a proxy object: `twisted.web.xmlrpc.Proxy`. The constructor for the object receives a URL: it must be an HTTP or HTTPS URL. When an XML-RPC service is described, the URL to that service will be given there.

Having a proxy object, one can just call the `callRemote` method, which accepts a method name and a variable argument list (but no named arguments, as these are not supported by XML-RPC). It returns a deferred, which will be called back with the result. If there is any error, at any level, the errback will be called. The exception will be the relevant Twisted error in the case of a problem with the underlying connection (for example, a timeout), `IOError` containing the status and message in the case of a non-200 status or a `xmlrpclib.Fault` in the case of an XML-RPC level problem.

```
from twisted.web.xmlrpc import Proxy
from twisted.internet import reactor

def printValue(value):
    print repr(value)
    reactor.stop()

def printError(error):
    print 'error', error
    reactor.stop()

proxy = Proxy('http://advogato.org/XMLRPC')
proxy.callRemote('test.sumprod', 3, 5).addCallbacks(printValue, printError)
reactor.run()
```

prints:

```
[8, 15]
```

7.4.5 Serving SOAP and XML-RPC simultaneously

`twisted.web.xmlrpc.XMLRPC` and `twisted.web.soap.SOAPPublisher` are both `Resources`. So, to serve both XML-RPC and SOAP in the one web server, you can use the `putChild` method of Resources.

The following example uses an empty `resource.Resource` as the root resource for a `Site`, and then adds `/RPC2` and `/SOAP` paths to it.

```
from twisted.web import soap, xmlrpc, resource, server
from TwistedQuotes import quoters
import os

quotefile = os.path.join(os.path.dirname(__file__), "quotes.txt")
quoter = quoters.FortuneQuoter([quotefile])

class XMLRPCQuoter(xmlrpc.XMLRPC):
    def xmlrpc_quote(self):
        return quoter.getQuote()

class SOAPQuoter(soap.SOAPPublisher):
    def soap_quote(self):
        return quoter.getQuote()

def main():
    from twisted.internet import reactor
    root = resource.Resource()
    root.putChild('RPC2', XMLRPCQuoter())
    root.putChild('SOAP', SOAPQuoter())
    reactor.listenTCP(7080, server.Site(root))
    reactor.run()

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

Source listing — *xmlAndSoapQuote.py*

Refer to *Twisted Web Development* (page 247) for more details about Resources.

Chapter 8

Woven

8.1 Woven

Note:

The Woven framework should not be used for new projects. The newer Nevow[1] framework, available as part of the Quotient[2] project, is a simpler framework with consistent semantics and better testing and is strongly recommended over Woven.

The Woven documentation below is maintained only for users with an existing Woven codebase.

8.1.1 Introduction

Woven is a Web Application Framework for building highly interactive web applications, written in Python. It separates HTML Templates from page-generation logic written in Python, and uses the Model View Controller (MVC) pattern to create dynamic HTML on the fly. Woven is higher level framework that depends on the Twisted Web package of the Twisted Framework.

8.1.2 Twisted Overview

Twisted is a framework written in Python for building network applications. A core design feature of Twisted is its asynchronous networking nature. Because of the high overhead and locking requirements of threads, Twisted chooses to instead use the highly efficient network event notification mechanisms provided by the OS in the form of the C `poll()` or `select()` calls. Twisted uses the Reactor pattern, which is an event-loop style of programming which facilitates asynchronous programming.

Asynchronous programming requires breaking your program into smaller function chunks that trigger an operation which may potentially take a long time (for example, a network request) and return. In order to continue the flow of your code when the operation has completed, you must register a "callback function" which will be called with the resulting data when it is ready. Twisted includes a generalized callback handling mechanism, Deferred, discussed in the Deferred execution paper[3].

[1] http://www.divmod.org/Home/Projects/Nevow/

[2] http://www.divmod.org/Quotient/

[3] http://www.twistedmatrix.com/documents/historic/2003/pycon/deferex/deferex.html

However, since writing Woven applications already involves breaking your functions into small, reusable Model-View-Controller components, Woven is able to handle the asynchronous nature of Twisted for you. By writing Models which provide data, potentially asynchronously, and Views which render data when it is ready, you are doing all that is required to act as a good citizen within the Twisted framework.

8.1.3 Twisted Web Object Publishing and Woven

Twisted includes a Web Server that handles HTTP requests and returns dynamic pages. It is useful when used in conjunction with Apache for serving static files such as Images. Apache can be set up to forward a specific URL via `ProxyPass` to the Twisted Web server. Twisted Web uses the concept of Object Publishing, similar to Zope, where there is a root Python Object below which all other Objects representing URLs are located.

When a request comes in to Twisted Web, Twisted Web splits the URL into segments and begins looking for Objects below the root by calling getChild tail-recursively. For example, if the URL `http://example.com/foo/bar/baz` is requested, Twisted splits this into the list of path segments `['foo', 'bar', 'baz']`. It then calls `root.getChild('foo')`, calls `getChild('bar')` on the result, calls `getChild('baz')` on the second result, and finally calls `render(request)` on the result of the final `getChild` call.

For more details about Twisted Web, see *Overview of Twisted Web* (page 13).

8.1.4 Smalltalk Model-View-Controller Overview

Originally developed for Smalltalk, the MVC design pattern is a flexible mechanism for creating both GUI and web application user interfaces. The primary advantage of the MVC pattern is separation of business logic from presentation details and provides a *loose coupling* between an application's Model (state) and View (presentation). All of this makes code reuse easier and enables a division of labor between application design and user interface design, albeit at the expense of a little extra work.

A "Model" is an object that represents or encapsulates an application's business logic or state. The model contains both data and business logic code, but does not contain presentation or rendering code.

A "View" is an object that contains presentation or rendering code, but does not contain business logic code.

Finally, a "Controller" is a dispatcher object that mediates flow between the Model and the View. In traditional Smalltalk MVC, the Controller is responsible for polling the Mouse and Keyboard and converting the user's actions (Click a button, for example) into high-level events (Change the Model data from 0 to 1, and redraw a View which represents this Model to the user).

8.1.5 Woven Model-View-Controller Overview

In Woven, a Model is a python object that holds the data that will eventually be rendered by a view object into a portion of an HTML page. Woven Models may be any Python object; Woven accomplishes this using "`IModel` Adapters":Components. Since different Python objects may act as "Containers" using different semantics, `IModel` adapters are required to allow Woven to address all Container objects uniformly. For example, Dictionaries are indexed using string keys. Lists are indexed using integer keys. Objects provide references to other objects using dot syntax (`foo.bar`), not square bracket syntax (`foo['bar']`). `IModel` Adapters are provided for all the basic Python types, Dictionaries, Lists, Integers, Floats, Strings, and Instances, and work behind the scenes to provide Model data to your Pages.

In Woven, a View is comprised of a Page object and many Widget objects. A Page object loads a template file (an XHTML file) that contains references to Widget objects (python objects that replace specific portions of the DOM created from the XHTML file). A single XHTML template references one Page object, but many Widget objects.

Widgets come in two flavors: local and global. Local Widgets are specific to only one template file, because the logic they perform is very specific, while global Widgets are general enough to replace placeholder content in many template files. Local Widgets are defined on the Page class, by defining a method with the prefix `wvupdate_`. Global Widgets are defined as subclasses of the Widget class.

In Woven, Page objects act as the default Controller object, similar to Servlets in Struts. ((please explain further?)) Since the Web is Request-Response and not event-driven like a Smalltalk application, the most basic event to be handled from the user is "URL Requested". The root Page object contains logic for locating the correct Page object to handle the request. Once this object is located, it handles the "url requested" event by rendering itself into HTML. Thus, the Page object acts as both the main View and Controller for this event.

However, unlike Struts, Woven also supports more Smalltalk-like Controller programming. With a Controller, it is possible to register a Python event handling function which will be called when a specific JavaScript event occurs in the Browser. These events are relayed to the server out-of-band (without causing the main page to refresh) and can be handled asynchronously. Thus, it is possible to program Web Applications which act more like traditional Smalltalk Desktop Applications.

Often in Woven, it is convenient for a single object to have both the View and Controller responsibilities, though this is not strictly necessary. It is only useful to split out the Controller logic from the View if the request argument handling is general enough to be reusable across multiple pages.

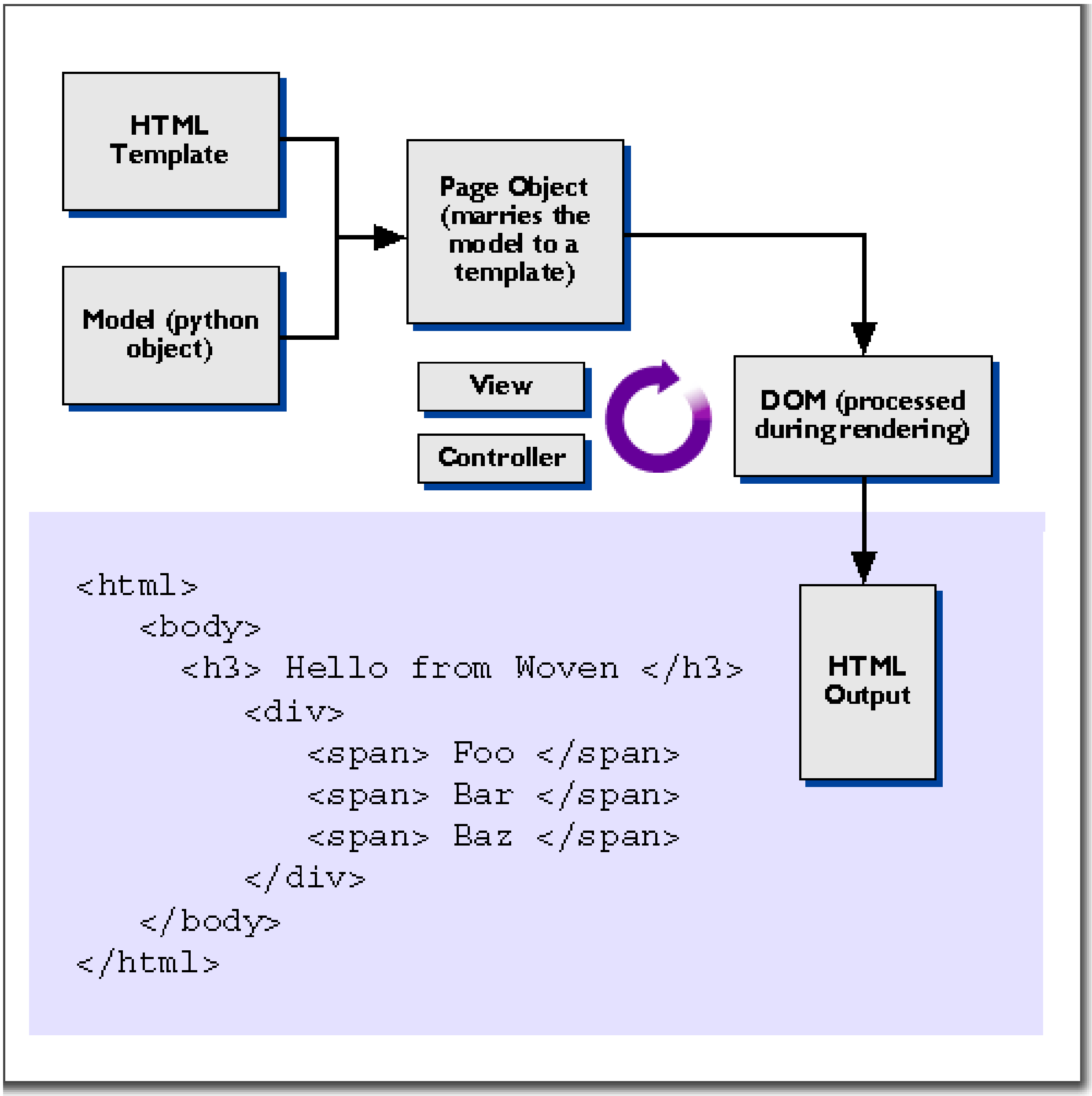
8.1.6 Overview of Woven Main Concepts

- *XHTML Templates* (page 271) must be valid XHTML documents. They provide a skeleton html page with placeholder content. Any element on the page which is destined for dynamic content must include special attributes to specify the name of model and view, like this: `<tag model="aModel" view="aView">placeholder text</tag>`
- *Model* (page 271) objects provide data for display in a web page.
- *View* (page 272) objects are given a DOM node created from the HTML template and model data, and are responsible for inserting the data into the node using the DOM api.
- *Controller* (page 272) objects accept input from the request and update the Model objects with the new input.
- *Page* (page 272) objects tie a model tree to a template, and provide view and controller factories to the template. This is the entry point of a web page built using Woven.
- *Rendering* (page 273) occurs when a web browser visits a Page. Woven recurses the template looking for nodes to handle, connects the correct Model object to a View and a Controller, and invokes them to render the node.
- *Further Concepts* (page 275) contains links to pages with more in depth information about Woven components.

8.1.7 In Depth Pages about Woven components

- *Twisted Web* (page 13)
- *Model* (page 283)
- *View* (page 289)
- *Controller* (page 296)

- *Page* (page 302)
- *LivePage* (page 300)
- *Form* (page 306)
- *Guard* (page 307)



8.1.8 Templates

Templates in Woven are XHTML documents, marked up with four special woven attributes:

- `model=` indicates which model object will be associated with this node.
- `view=` indicates which view object will be associated with this node.
- `controller=` indicates which controller object will be associated with this node.
- `pattern=` marks a node so it may be located by the View code, without the view knowing where the pattern= node is located or what style the pattern node contains.

HTML Template example

```
<html>
  <body>
    <h3 model="name" view="Text" />
    <div model="stuff" view="List">
      <span pattern="listItem" view="Text" />
    </div>
  </body>
</html>
```

8.1.9 Models

Model objects are arranged in a tree. Each Woven page has a model tree with exactly one root. All data required for display by a Woven page is made available through named sub-models below this root.

Any Python object may be used as a Woven model.

An example of a model tree built using simple python data structures, Dictionaries, Lists, and Strings:

```
model = {
  'name': 'Welcome to Woven',
  'stuff': [
    'Foo',
    'Bar',
    'Baz'
  ]
}
```

Each model in this tree has a submodel path which we can use to refer to it.

- The main dictionary is named `/`
- The name string is named `/name`
- The stuff list is named `/stuff`
- The first element of the stuff list is named `/stuff/0`
- The second element of the stuff list is named `/stuff/1`
- The third element of the stuff list is named `/stuff/2`

8.1.10 Views

View objects are constructed each time a `view=` directive is encountered.

If `view="Text"` is specified on a `Node`, an instance of the `Text` class will be constructed to handle the node.

An example View widget which simply inserts the model data into the DOM as a text node:

```
class Text(view.View):
    def generate(self, request, node, model):
        data = str(model.getData())
        newTextNode = request.d.createTextNode(data)
        node.appendChild(newTextNode)
        return node
```

The node that is returned from the generate method replaces the template node in the final HTML output.

> **Note:**In the above case, the same node that was passed in was returned, after being changed ("mutated").

The `View` object should return a DOM node that has been populated with the given model data.

8.1.11 Controllers

Controllers are responsible for accepting input from the user and making appropriate changes to the model.

They are also responsible for ensuring that the data submitted by the user is valid before changing the model.

Very few applications need the flexibility of separate Controller objects. Often, is is more convenient and clear to place the Controller logic in the View, where the View can make sure it has the latest data before rendering itself.

An example of a Controller which verifies the user's input before committing it to the Model:

```
class NewName:
    def handle(self, request):
        newName = request.args.get("newname", None)
        if newName is None:
          ## The user did not submit the form; do not do anything.
          return
        if newName == "":
          self.view.setError("Sorry, you didn't enter a name.")
        else:
          self.model.setData(newName)
          ## Tell the model that we are done making changes to it,
          ## and Views that rely upon this model should rerender.
          self.model.notify({'request': request})
```

8.1.12 Pages

Pages are the entry point into a Woven application. The Page object accepts a request to render a web page from `twisted.web` and drives the page rendering process by parsing the template, locating Model objects in the tree, and invoking Controller and View objects.

Page is a subclass of Resource, the `twisted.web` class representing an individual URL. Resource instances can be hooked up to the twisted.web HTTP server in several ways. The simplest way to hook up a Resource to a web URL is to start a static twisted.web server, which will serve files out of the given directory:

```
% mktap web --path /Users/dsp/Sites
% twistd -nf web.tap
```

If you visit the URL `http://localhost:8080/`, you will see the contents of the directory you specified. To create a URL which will be served by an instance of `Page`, create a Python script. In this script, instantiate a `Page` instance, passing it a `Model` and a `Template`, and assign it to a variable named `resource`:

```
from twisted.web.woven import page

model = {'name': 'Welcome to Woven',
         'stuff': ['Foo', 'Bar', 'Baz']}

template = """<html>
  <body>
      <h3 model="name" view="Text" />
      <div model="stuff" view="List">
        <p pattern="listItem" view="Text" />
      </div>
    </body>
  </html>
"""

resource = page.Page(model, template=template)
```

Name this script `test.rpy` and place it in the directory served by `twisted.web`. Then visit the URL `http://localhost:8080/test.rpy` with your web browser, and you should see a page with the HTML-formatted model data.

Page rendering process

When Woven renders a page, it first constructs a DOM (Document Object Model) which represents the template in memory using Python objects.

Woven then examines the DOM using a pre-order traversal looking for nodes with woven directives (nodes with `model=`, `view=`, or `controller=` attributes).

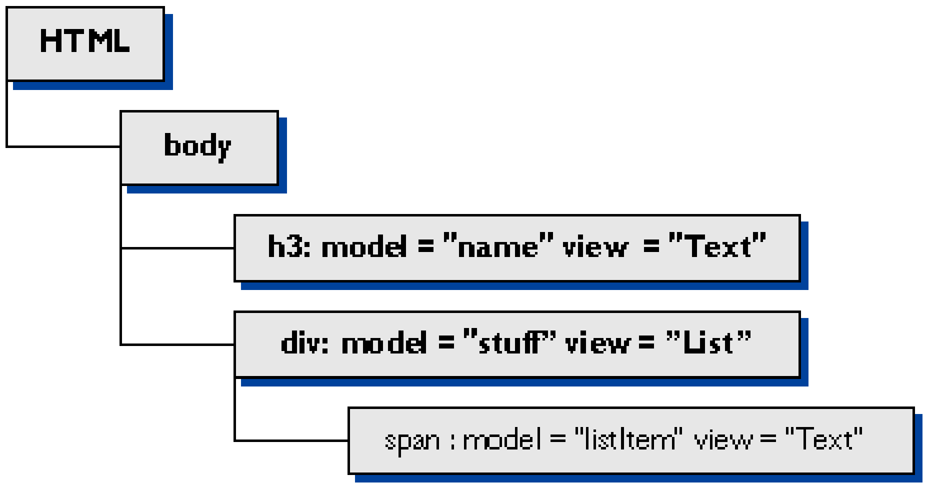
When a directive node is encountered, Woven locates/constructs Model, View, and Controller objects to handle this node.

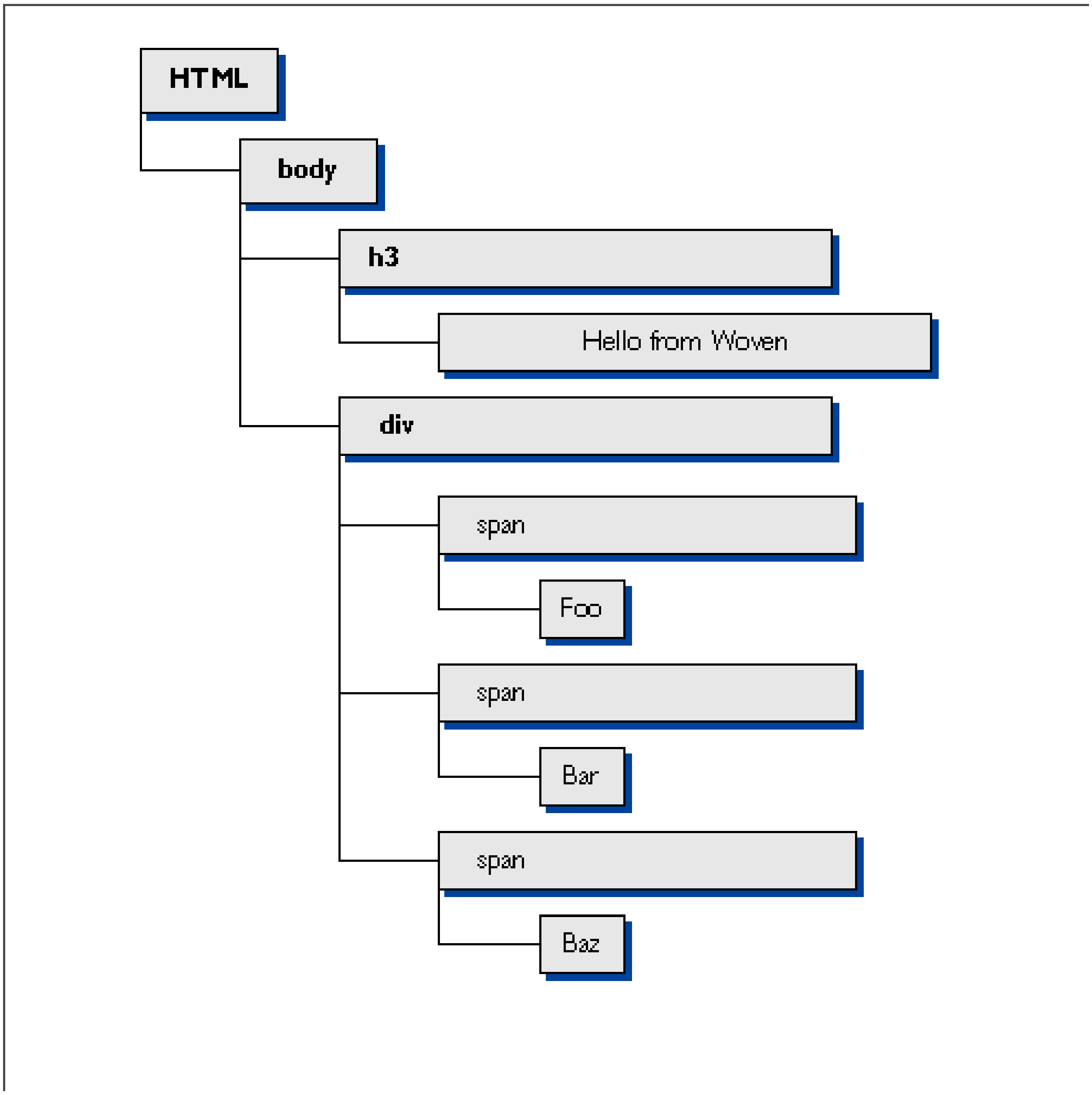
The Controller is then triggered by calling handle. The controller may take any action required of it to update the Model data, or may do nothing.

The View is then rendered by calling generate. The View object is passed a DOM node and a reference to the Model. The view then manipulates the DOM node, placing the Model data in it.

The DOM returned from the View is then traversed, looking for directives to handle.

When the entire DOM has been traversed and mutated, the DOM is converted to HTML and sent to the browser.


HTML
body
h3: model = "name" view = "Text"
div: model = "stuff" view = "List"
span : model = "listItem" view = "Text"



8.1.13 Further Reading

Twisted Web (page 13) is the Object-publishing web server woven uses to glue HTTP requests to Page rendering.

Page (page 302) objects are the IResource implementors in the Woven framework. They represent URL segments and drive the Woven template rendering process. They also contain convenient methods for specifying Page trees.

Model (page 283) objects provide data to Woven pages. Woven includes IModel adapters for the basic Python types and makes it easy for you to write your own Model classes and IModel adapters for your already-existing classes.

View (page 289) objects insert Model data into the DOM provided by the Template. They use DOM syntax for HTML generation, and have convenient syntax for locating and copying `pattern=` nodes abstractly.

Controller (page 296) objects accept input from the request and update the Model data. Controller objects are optional; often your Page or View instances can contain the controller logic and still make sense. Controllers are for cases which are general enough to warrant validation and commit logic.

LivePage (page 300) objects allow you to build DHTML behaviors into your Woven pages using pure server-side Python. It includes code for asynchronously forwarding client-side JavaScript events to the server without refreshing the page, and sending JavaScript to a page from the server after it has already been loaded.

Form (page 306) is a woven module that makes it easy to create HTML forms from existing Python methods which take keyword arguments. It also supports basic input validation.

Guard (page 307) is a woven module that allows you to wrap your Page instances with login pages, to prevent unauthorized users from accessing them.

8.2 PicturePile: a tutorial Woven application

Note:

The PicturePile tutorial illustrates a simple Woven web application. However, the Woven framework should not be used for new projects. The newer Nevow[4] framework, available as part of the Quotient[5] project, is a simpler framework with consistent semantics and better testing and is strongly recommended over Woven.

The tutorial is maintained only for users with an existing Woven codebase.

To illustrate the basic design of a Woven app, we're going to walk through building a simple image gallery. Given a directory of images, it will display a listing of that directory; when a subdirectory or image is clicked on, it will be displayed.

To begin, we write an HTML template for the directory index, and save it as directory-listing.html:

```
<html>
  <head>
    <title model="title" view="Text">Directory listing</title>
  </head>
  <body>
    <h1 model="title" view="Text"></h1>
    <ul model="directory" view="List">
      <li pattern="listItem"><a view="Anchor" /></li>
      <li pattern="emptyList">This directory is empty.</li>
    </ul>
  </body>
</html>
```

[4] http://www.divmod.org/Home/Projects/Nevow/
[5] http://www.divmod.org/Quotient/

The main things that distinguish a Woven template from standard XHTML are the `model`, `view`, and `pattern` attributes on tags. Predictably, `model` and `view` specify which model and view will be chosen to fill the corresponding node. The `pattern` attribute is used with views that have multiple parts, such as List. This example uses two patterns `List` provides; `listItem` marks the node that will be used as the template for each item in the list, and `emptyList` marks the node displayed when the list has no items.

Next, we create a `Page` that will display the directory listing, filling the template above (after a few imports):

```
import os
from twisted.application import service, internet
from twisted.web.woven import page
from twisted.web import server
from twisted.web import microdom
from twisted.web import static

class DirectoryListing(page.Page):

    templateFile = "directory-listing.xhtml"
    templateDirectory = os.path.split(os.path.abspath(__file__))[0]

    def initialize(self, *args, **kwargs):
        self.directory = kwargs['directory']

    def wmfactory_title(self, request):
        """Model factory for the title.

        This method will be called to create the model to use when
        'model="title"' is found in the template.
        """
        return self.directory

    def wmfactory_directory(self, request):
        """Model factory for the directory.

        This method will be called to create the model to use when
        'model="directory"' is found in the template.
        """
        files = os.listdir(self.directory)
        for i in xrange(len(files)):
            if os.path.isdir(os.path.join(self.directory,files[i])):
                files[i] = files[i] + '/'
        return files

    def getDynamicChild(self, name, request):
        # Protect against malicious URLs like '..'
        if static.isDangerous(name):
            return static.dangerousPathError
```

```
        # Return a DirectoryListing or an ImageDisplay resource, depending on
        # whether the path corresponds to a directory or to a file
        path = os.path.join(self.directory,name)
        if os.path.exists(path):
            if os.path.isdir(path):
                return DirectoryListing(directory=path)
            else:
                return ImageDisplay(image=path)
```

Due to the somewhat complex inheritance hierarchy in Woven's internals, a lot of processing is done in the `__init__` method for `Page`. Therefore, a separate `initialize` method is provided so that one can easily access keyword args without having to disturb the internal setup; it is called with the same args that `Page.__init__` receives.

The `templateFile` attribute tells the Page what file to load the template from; in this case, we will store the templates in the same directory as the Python module. The `wmfactory` (short for Woven Model Factory) methods return objects to be used as models; In this case, `wmfactory_title` will return a string, the directory's name, and `wmfactory_directory` will return a list of strings, the directory's content.

Upon rendering, Woven will scan the template's DOM tree for nodes to fill; when it encounters one, it gets the model (in this case by calling methods on the Page prefixed with `wmfactory_`), then creates a view for that model; this page uses standard widgets for its models and so contains no custom view code. The view fills the DOM node with the appropriate data. Here, the view for `title` is `Text`, and so will merely insert the string. The view for `directory` is `List`, and so each element of the list will be formatted within the '<ul>'. Since the view for list items is Anchor, each item in the list will be formatted as an `<a>` tag.

So, for a directory "Images" containing "foo.jpeg", "baz.png", and a directory "MoreImages", the rendered page will look like this:

```
<html>
  <head>
    <title>/Users/ashort/Pictures</title>
  </head>
  <body>
    <h1>/Users/ashort/Pictures</h1>
    <ul>
      <li>
        <a href="foo.jpeg">foo.jpeg</a>
      </li>
      <li>
        <a href="baz.png">baz.png</a>
      </li>
      <li>
        <a href="MoreImages/">MoreImages/</a>
      </li>
    </ul>
  </body>
</html>
```

As you can see, the nodes marked with `model` and `view` are replaced with the data from their models, as formatted by their views. In particular, the List view repeated the node marked with the `listItem` pattern for each item in the list.

For displaying the actual images, we use this template, which we save as image-display.html:

```
<html>
  <head>
    <title model="image" view="Text">Filename</title>
  </head>
  <body>
    <img src="preview" />
  </body>
</html>
```

And here is the definition of `ImageDisplay`:

```
from twisted.web import static

class ImageDisplay(page.Page):

    templateFile="image-display.xhtml"

    def initialize(self, *args, **kwargs):
        self.image = kwargs['image']

    def wmfactory_image(self, request):
        return self.image

    def wchild_preview(self, request):
        return static.File(self.image)
```

Instead of using `getDynamicChild`, this class uses a `wchild_` method to return the image data when the `preview` child is requested. `getDynamicChild` is only called if there are no `wchild_` methods available to handle the requested URL.

Finally, we create a webserver set to start with a directory listing, and connect it to a port. We will tell this Site to serve a DirectoryListing of a directory named "Pictures" in our home directory:

```
rootDirectory = os.path.expanduser("~/Pictures")
site = server.Site(DirectoryListing(directory=rootDirectory))
application = service.Application("ImagePool")
parent = service.IServiceCollection(application)
internet.TCPServer(8088, site).setServiceParent(parent)
```

And then start the server by running the following command-line: `twistd -noy picturepile.py`.

8.2.1 Custom Views

Now, let's add thumbnails to our directory listing. We begin by changing the view for the links to "thumbnail":

```
<html>
  <head>
    <title model="title" view="Text">Directory listing</title>
  </head>
  <body>
    <h1 model="title" view="Text"></h1>
    <ul model="directory" view="List">
      <li pattern="listItem"><a view="thumbnail" /></li>
      <li pattern="emptyList">This directory is empty.</li>
    </ul>
  </body>
</html>
```

Woven doesn't include a standard "thumbnail" widget, so we'll have to write the code for this view ourselves. (Standard widgets are named with initial capital letters; by convention, custom views are named like methods, with initial lowercase letters.)

The simplest way to do it is with a `wvupdate_` (short for Woven View Update) method on our DirectoryListing class:

```
def wvupdate_thumbnail(self, request, node, data):
    a = microdom.lmx(node)
    a['href'] = data
    if os.path.isdir(os.path.join(self.directory,data)):
        a.text(data)
    else:
        a.img(src=(data+'/preview'),width='200',height='200').text(data)
```

When the `thumbnail` view is requested, this method is called with the HTTP request, the DOM node marked with this view, and the data from the associated model (in this case, the name of the image or directory). With this approach, we can now modify the DOM as necessary. First, we wrap the node in `lmx`, a class provided by Twisted's DOM implementation that provides convenient syntax for modifying DOM nodes; attributes can be treated as dictionary keys, and the `text` and `add` methods provide for adding text to the node and adding children, respectively. If this item is a directory, a textual link is displayed; else, it produces an `IMG` tag of fixed size.

8.2.2 Simple Input Handling

Limiting thumbnails to a single size is rather inflexible; our app would be nicer if one could adjust it. Let's add a list of thumbnail sizes to the directory listing. Again, we start with the template:

```
<html>
<head>
  <title model="title" view="Text">Directory listing</title>
</head>
<body>
  <h1 model="title" view="Text"></h1>
  <form action="">
    Thumbnail size:
```

```
      <select name="thumbnailSize" onChange="submit()" view="adjuster">
        <option value="400">400x400</option>
        <option value="200">200x200</option>
        <option value="100">100x100</option>
        <option value="50">50x50</option>
      </select>
    </form>
    <ul model="directory" view="List">
      <li pattern="listItem"><a view="thumbnail" /></li>
      <li pattern="emptyList">This directory is empty.</li>
    </ul>
  </body>
</html>
```

Source listing — *directory-listing3.html*

This time, we add a form with a list of thumbnail sizes named `thumbnailSize`: we want the form to reflect the selected option, so we place an `adjuster` view on the `select` tag that looks for the right `option` tag and puts `selected=1` on it (the default size being 200):

```
def wvupdate_adjuster(self, request, widget, data):
    size = request.args.get('thumbnailSize',('200',))[0]
    domhelpers.locateNodes(widget.node.childNodes,
                           'value', size)[0].setAttribute('selected', '1')
```

`request.args` is a dictionary, mapping argument names to lists of values (since multiple HTTP arguments are possible). In this case, we only care about the first argument named `thumbnailSize`. `domhelpers.locateNodes` is a helper function which, given a list of DOM nodes, a key, and a value, will search each tree and return all nodes that have the requested key-value pair.

Next, we modify the `thumbnail` view to look at the arguments from the HTTP request and use that as the size for the images:

```
def wvupdate_thumbnail(self, request, node, data):
    size = request.args.get('thumbnailSize',('200',))[0]
    a = microdom.lmx(node)
    a['href'] = data
    if os.path.isdir(os.path.join(self.directory,data)):
        a.text(data)
    else:
        a.img(src=(data+'/preview'),width=size,height=size).text(data)
```

8.2.3 Sessions

A disadvantage to the approach taken in the previous section is that subdirectories do receive the same thumbnail sizing as their parents; also, reloading the page sets it back to the default size of 200x200. To remedy this, we need a way to store data that lasts longer than a single page render. Fortunately, twisted.web provides this in the form of a Session object. Since only one Session exists per user for all applications on the server, the Session object is Componentized, and each application adds adapters to contain their own state and behaviour, as explained in the *Components* (page 143) documentation. So, we start with an interface, and a class that implements it, and registration of our class upon Session:

```
class IPreferences(components.Interface):
    pass

class Preferences(components.Adapter):
    __implements__ = IPreferences

components.registerAdapter(Preferences, server.Session, IPreferences)
```

We're just going to store data on this class, so no methods are defined.

Next, we change our view methods, `wvupdate_thumbnail` and `wvupdate_adjuster`, to retrieve their size data from the Preferences object stored on the Session, instead of the HTTP request:

```
    def wvupdate_thumbnail(self, request, node, data):
        prefs = request.getSession(IPreferences)
        size = getattr(prefs, 'size','200')
        a = microdom.lmx(node)
        a['href'] = data
        if os.path.isdir(os.path.join(self.directory,data)):
            a.text(data)
        else:
            a.img(src=(data+'/preview'),width=size,height=size).text(data)

    def wvupdate_adjuster(self, request, widget, data):
        prefs = request.getSession(IPreferences)
        size = getattr(prefs, 'size','200')
        domhelpers.locateNodes(widget.node.childNodes,
                              'value', size)[0].setAttribute('selected', '1')
```

Controllers

Now we turn to the question of how the data gets into the session in the first place. While it is possible to to place it there from within the `wvupdate_` methods, since they both have access to the HTTP request, it is desirable at times to separate out input handling, which is what controllers are for. So, we add a `wcfactory_` (short for Woven Controller Factory) method to DirectoryListing:

```
    def wcfactory_adjuster(self, request, node, model):
        return ImageSizer(model, name='thumbnailSize')
```

ImageSizer is a controller. It checks the input for validity (in this case, since it subclasses `Anything`, it merely ensures the input is non-empty) and calls `handleValid` if the check succeeds; in this case, we retrieve the Preferences component from the session, and store the size received from the form upon it:

```
class ImageSizer(input.Anything):
    def handleValid(self, request, data):
```

```
        prefs = request.getSession(IPreferences)
        prefs.size = data
```

Finally, we must modify the template to use our new controller. Since we are concerned with the input from the `<select>` element of the form, we place the controller upon it:

```
    <html>
    <head>
      <title model="title" view="Text">Directory listing</title>
    </head>
    <body>
      <h1 model="title" view="Text"></h1>
      <form action="">
        Thumbnail size:
        <select name="thumbnailSize" onChange="submit()" view="adjuster"
          controller="adjuster">
          <option value="400">400x400</option>
          <option value="200">200x200</option>
          <option value="100">100x100</option>
          <option value="50">50x50</option>
        </select>
      </form>
      <ul model="directory" view="List">
        <li pattern="listItem"><a view="thumbnail" /></li>
        <li pattern="emptyList">This directory is empty.</li>
      </ul>
    </body>
  </html>
```

Source listing — *directory-listing4.html*

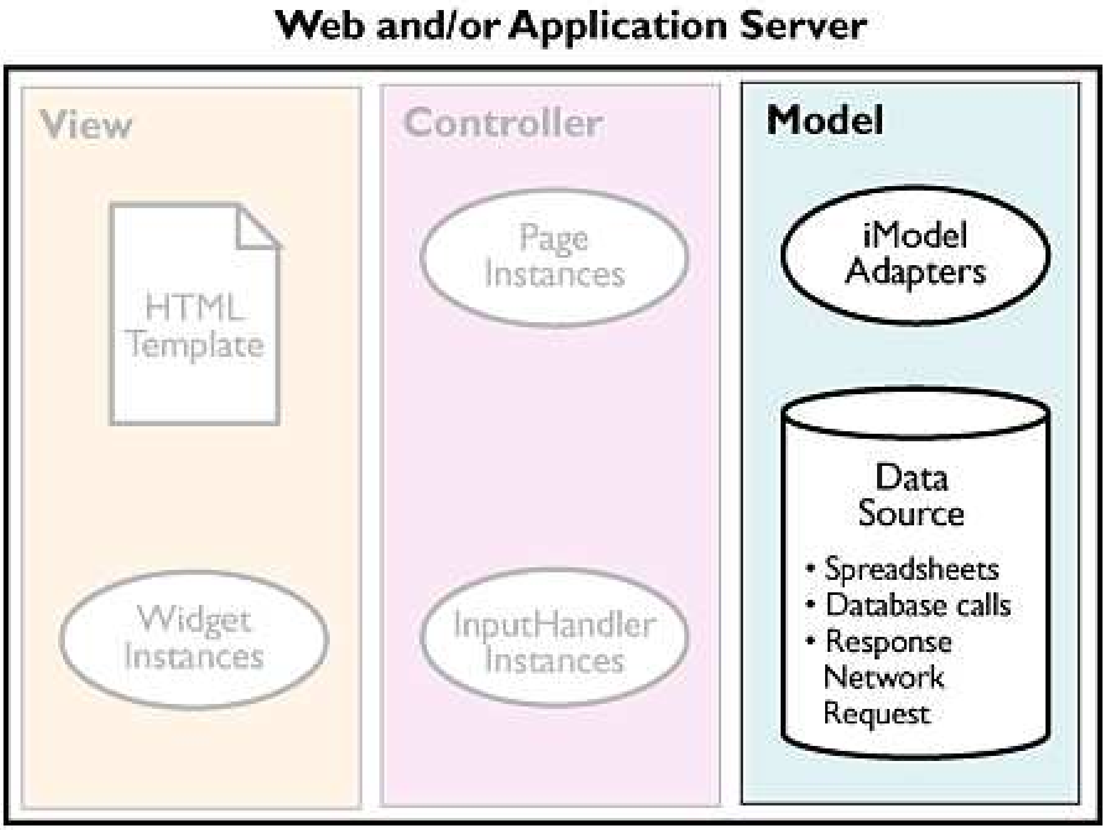
Now, the selected size will be remembered across subdirectories and page reloads.

8.3 Model In Depth

Note:
This HOWTO documents the Model class, part of the Woven framework. The Woven framework should not be used for new projects. The newer Nevow[6] framework, available as part of the Quotient[7] project, is a simpler framework with consistent semantics and better testing and is strongly recommended over Woven.

The Woven documentation below is maintained only for users with an existing Woven codebase.

[6]http://www.divmod.org/Home/Projects/Nevow/
[7]http://www.divmod.org/Quotient/



Model objects provide data to View objects as a Page is being rendered.

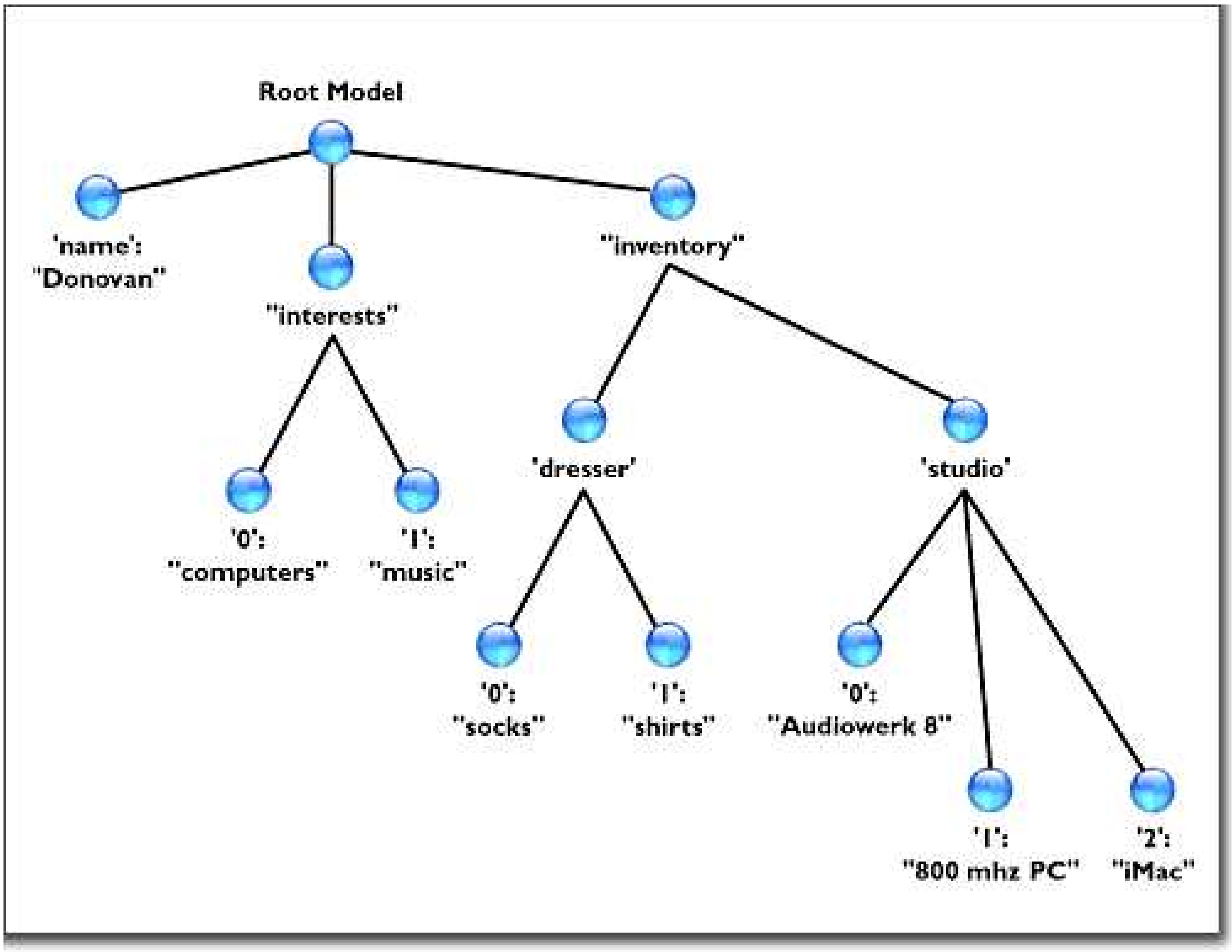
8.3.1 Main Concepts

- Root Models are the data entry point for every woven Page. All Model data for display on a Page should be made available through this Root Model. Described further in the *Page section* (page 303).
- *Submodel Paths* (page 285) allow Woven to locate the correct Model data for a node.
- *The Model Stack* (page 286) is how Woven keeps track of which Model object is currently "in scope". Instead of specifying `model=` attributes in the Template with "absolute submodel paths", you can specify a model relative to the top of the "Model stack" with a "relative submodel path".
- *IModel Adapters* (page 287) allow you to write wrappers for existing objects. Subclassing a base Model in the `models` module will make writing an IModel Adapter easier.
- *Model Factories* (page 288) allow you to produce Model objects on demand with a Python method.

8.3.2 Submodel Paths

Each Model Woven has access to in the tree has a "submodel path". Submodel paths start at the Root Model and specify each segment Woven must follow to locate the Model. Submodel paths are slash-separated strings similar to filesystem paths. For the basic Python container types, Dictionaries and Lists, a submodel path segment is simply the key into the container. Given the model:

```
model = {'name': "Donovan",
         'interests': ["Computers", "Music"],
         'inventory': {'dresser': ['socks','shirts'],
                       'studio': ['Audiowerk8', '800mhz PC', 'iMac'],
                      }
         }
```



The following submodel paths are valid:

- / specifies the Root Model, a Dictionary

- `/name` specifies the `name` entry in the Dictionary, a String
- `/interests` specifies the `interests` entry in the Dictionary, a List
- `/interests/0` specifies the first element of the `interests` list, a String
- `/interests/1` specifies the second element of the `interests` list, a String
- `/inventory/dresser/0` specifies the first element of the `dresser` list in the `inventory` dictionary, a String

etc...

When woven encouters a `model=` directive on a node, it will look up the model and pass it to the View object that will render the node:

```
<html>
    <body>
        <h3 model="/interests/3" view="Text" />
    </body>
</html>
```

8.3.3 The Model Stack and Relative Submodel Paths

While "absolute model paths" are useful for specifying exactly which Model data you want associated with a node, the more frequent use case is to specify a "relative model path" which is a path relative to the Model currently on top of the "Model stack". Relative model paths are easy to distinguish because they do not begin with a slash.

When Woven encounters a node with a `model=` attribute, it looks up the Model object and places it on top of the "Model stack". During the processing of this node and all of the node's child nodes, this Model object remains on the top of the stack. Once all child nodes have completed processing, it is popped off of the Model stack.

This means that child nodes can refer to elements of the Model on top of the Model stack with relative submodel paths. For example, we may wish to render the "interests" list from the above example as two separate HTML elements. To do so, we first place the "interests" list on top of the Model stack, and then refer to elements of this list:

```
<html>
  I am interested in:
  <div model="interests" view="None">
    <p>First thing: <span model="0" view="Text" /></p>
    <p>Second thing: <span model="1" view="Text" /></p>
  </div>
</html>
```

In this case, the "interests" list was in scope for the duration of the <div> tag, and the individual interest strings were in scope for the duration of the individual <span> tags.

The List widget uses this Stack concept to operate on DOM nodes abstractly, without knowing or caring what directives will occur when the child nodes it returns are handled. We can also use the familiar `.` and `..` concepts from unix shell syntax to refer to Models:

```
<html>
  <div model="interests" view="List">
    <h3 pattern="listHeader" model="../name" view="Text" />>
    <p pattern="listItem" view="Text" />
    <h6 pattern="listFooter" model="." view="Text" />
  </div>
<html>
```

The List widget makes copies of the `pattern` nodes without knowing or caring which directives have been placed on them, or how many children are contained within the node. It then simply sets the `model=` attribute of each of the nodes to the correct index into the list. More about pattern directives is available in the *Views* (page 289) section.

In the above example, even though the `interests` list had been placed on the Model stack, we were able to access the `name` string without knowing its absolute path by using the relative path `../name`, and we were able to render the `interests` list with a different View Widget using the relative path `..`

The output from generating the above HTML will look like this:

```
<html>
  <div>
    <h3>Donovan</h3>
    <p>Computers</p>
    <p>Music</p>
    <h6>['Computers', 'Music']</h6>
    </div>
</html>
```

8.3.4 IModel Adapters

The IModel interface is documented in `twisted.web.woven.interfaces.IModel`. It describes the interfaces Models must implement in order to participate in the Woven Model stack. If you are inheriting from `twisted.web.woven.model.Model`, most of these interfaces will be implemented for you.

The interfaces that we will be most interested in implementing are those that are designed to be overridden for customization, getData and setData.

For example, we may wish to create a wrapper for some data which we will retrieve out of a SQL database. To do so, we create a subclass of Model:

```
class DatabaseJunk(model.Model):
    def getData(self, request):
        someCursor.execute("select * from foo")
        return someCursor.fetchall()

    def getSubmodel(self, request, name):
        row = self.getData(request)[int(name)]
        return RowUpdater(row)

class RowUpdater(model.Model):
    def __init__(self, id):
        self.row = row
```

```
    def getData(self, request):
        return self.row

    def setData(self, request, data):
        someCursor.execute(
        "update foo set bar=%s, baz=%s where id = %s",
        (data[0], data[1], self.row[0]))
```

The result of getData must be an IModel implementor, or may be a Deferred. Thus you may use the IModel interface to produce data from an adbapi call, a pb call, etc. When the data returned is a Deferred, Woven will pause rendering of the current node until the data is available.

8.3.5 Registering an IModel adapter for a class

Woven makes use of the twisted component system. Components, which are discussed in the *Components* (page 143) section, allow classes to declare that they implement a specific Interface for another class. This is useful if you already have classes in which you store data, and wish to create thin IModel adapter wrappers around them:

```
class MyData:
    def __init__(self, something=""):
        self.something = something

class MyDataModel(models.MethodModel):
    ## When the MyDataModel adapter is wrapped around an instance
    ## of MyData, the original MyData instance will be stored in 'orig'
    def wmfactory_something(self, request):
        return self.orig.something

from twisted.python import components
from twisted.web.woven import interfaces

components.registerAdapter(MyDataModel, MyData, interfaces.IModel)
```

8.3.6 Model Factories

Using a separate Model class for each individual piece of data in the system makes sense when you are able to generalize your Model classes enough so they are reusable. However, it is often easier, especially if you need to perform highly varied SQL calls to produce your data, to use a Model which supports Model factories.

There are two ways to use Model factories. The first is to have a separate Model class which subclasses `model.MethodModel`. The second is to simply not pass any Model at all to the Page instance, in which case the Page itself will act as a MethodModel.

MethodModel classes should provide methods prefixed with `wmfactory_`, which will be called when the directive `model=` is present in a template. For example, given the node `<div model="foo" />`, a method named `wmfactory_foo` will be called:

```
class MyModel(model.MethodModel):
```

```
    def wmfactory_foo(self, request):
        return ['foo', 'bar', 'baz']
```

If you did not pass any Model object when you created your Page instance, the Page class will act as a MethodModel. Thus, you can place your `wmfactory_` methods directly on your Page subclass:

```
class MyPage(page.Page):
    def wmfactory_foo(self, request):
        return ['foo', 'bar', 'baz']
```

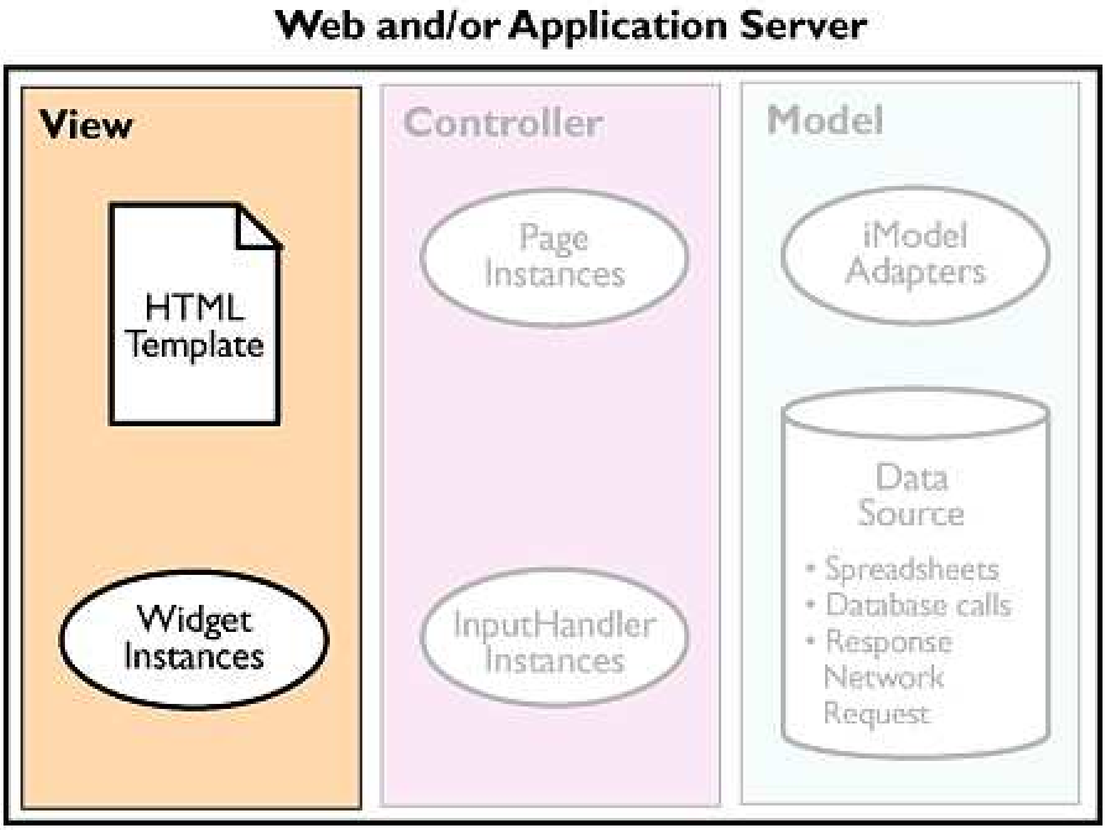
Model factories are a useful way to write some Python code which generates your page model data, for pages which need to look up or calculate data in some way.

8.4 View In Depth

Note:
This HOWTO documents the View class, part of the Woven framework. The Woven framework should not be used for new projects. The newer Nevow[8] framework, available as part of the Quotient[9] project, is a simpler framework with consistent semantics and better testing and is strongly recommended over Woven.

The Woven documentation below is maintained only for users with an existing Woven codebase.

[8] http://www.divmod.org/Home/Projects/Nevow/
[9] http://www.divmod.org/Quotient/



View objects are given a Model and a template DOM node, and use the DOM api to insert the given Model data into the DOM. Views are where you manipulate the HTML, in the form of DOM, which will be sent to the web browser.

8.4.1 Main Concepts

- *View factories* (page 291) provide the glue from a DOM node with a `'view='` directive to an instance of a View class.
- *generate* (page 291) is the method which is called on the View instance to render a node.
- *Widgets* (page 292) are views that have convenient syntax for rendering a View.
- *lmx* (page 293) is a much more convenient API for constructing DOM nodes programatically than the actual DOM API.

- *wvupdate_* (page 295) methods provide a convenient way to customize a generic Widget's rendering process without writing an inconvenient amount of boilerplate.
- *The View stack* (page 295) allows your View classes to expose private subviews which are only visible while your View is in scope on the stack.

8.4.2 View factories

View factories provide the glue between a `view=` directive on a DOM node and a View instance. When a DOM node with a `view=` directive is encountered, Woven looks for a corresponding `wvfactory_` method on your Page instance. A View factory is required to return an object which implements the interface `IView`:

```
class MyView(view.View):
  def generate(self, request, node):
      return request.d.createTextNode("Hello, world!")

class MyPage(page.Page):
    def wvfactory_foo(self, request, node, model):
        return MyView(model)
```

A View factory should almost always construct the View with the Model object it is passed. The exception to this rule is when the View is designed to render data which is not available in the Model tree, such as data which is obtained from the request or from a session object:

```
class MyPage(page.Page):
    def wvfactory_currentPageName(self, request, node, model):
        return widgets.Text(request.prepath[-1])
```

Note that if the Model the View is constructed with is not the Model which was passed in to the factory, Woven will notice this and place the new Model data on the Model stack.

You may set View factories programatically on a generic Page instance after it has been constructed. The first View factory example could be written:

```
class MyView(view.View):
    def generate(self, request, node):
        return request.d.createTextNode("Hello, world!")

resource = page.Page()
resource.setSubviewFactory("foo", MyView)
```

8.4.3 generate

The generate method is the most important method in the IView interface. It is the entry point from the Woven framework into your custom Python View code. When your View instance was constructed, it was passed a Model as the first argument. This is the Model data which generate should insert into the DOM. generate is passed the request and a template DOM node, and must return a DOM node, which will replace the template in the DOM tree:

```
class MyView(view.View):
    def generate(self, request, node):
        return request.d.createTextNode("Hello, world!")
```

Note that the current DOM Document object is available as `request.d`. You should use this document object as a text node and element factory, so the details about the underlying DOM implementation remain hidden.

Often, it is incredibly useful to use the incoming template node as a "skin" for your Views. In the simplest form, this involves simply adding nodes to the incoming template node and returning it from generate:

```
class MySkinnedView(view.View):
      def generate(self, request, node):
          modelData = self.getData()
          newNode = request.d.createTextNode(str(modelData))
          node.appendChild(newNode)
          return node
```

However, Woven also supports the concept of "pattern=" nodes, nodes which are marked in the template with a given "pattern=" directive so they may be located by the View abstractly. To support this, Woven contains a View subclass called Widget, which provides a far more convenient API for rendering your Views.

8.4.4 Widgets

Rendering Views is such an important part of developing a Woven application that it needs to be as convenient as possible. To support reducing the amount of boilerplate required to write a new View, the View subclass Widget was created. When subclassing Widget, simply override setUp instead of generate. setUp differs from generate in that it is passed a reference to the Model data, not the Model wrapper, and may simply mutate the template DOM node in place without having to worry about returning anything:

```
class MyWidget(widgets.Widget):
    def setUp(self, request, node, data):
        newNode = request.d.createTextNode(str(data))
        node.appendChild(newNode)
```

Widget also supports a very useful and convenient method called getPattern which allows you to locate nodes in the template node which have a `pattern=` attribute on them, regardless of where they are in the template, what style the node is, or how many children the node has:

```
class MyPatternWidget(widgets.Widget):
    def setUp(self, request, node, data):
        if data > 10:
            newNode = self.getPattern("large")
        else:
            newNode = self.getPattern("small")
        node.appendChild(newNode)
```

This widget could be used with the following template to abstractly allow the designer to style elements which are larger than 10:

```
<div model="listOfIntegers" view="List">
  <div pattern="listItem" view="MyPatternWidget">
      <span pattern="large" style="background-color: red" view="Text" />
      <span pattern="small" style="background-color: blue" view="Text" />
  </div>
</div>
```

Notice how the Widgets chain themselves to create the final page; the List widget makes copies of the pattern node which have `view="MyPatternWidget"` on them; the `MyPatternWidget` widget makes copies of the `pattern="large"` or `pattern="small"` nodes which have `view="Text"` directives on them; and the Text widget inserts the actual model data from the list into the innermost `<span>` element.

Widgets, along with the DOM api and the getPattern helper method, provide a powerful way for you to write view logic in Python without knowing or caring what type of HTML nodes are in your Template.

8.4.5 lmx

Generating View structure using the DOM is very useful for separating the Template from the actual logic which structures the View. However, if you need to do a large amount of HTML generation in Python, it becomes very cumbersome quickly. lmx is a lightweight wrapper around DOM nodes that allows you to quickly and easily build large HTML structures from Python:

```
from twisted.web.microdom import lmx

class LMXWidget(widgets.Widget):
    def setUp(self, request, node, data):
        l = lmx(node)
        for color in data:
            l.div(style=
                "width: 2in; height: 1in; background-color: %s" % color)
```

When an lmx instance is wrapped around a DOM node, calling a method on the lmx instance creates a new DOM node inside a new lmx instance. The new DOM node will have the same tag name as the name of the method that was called, and an attribute for each keyword argument that was passed to the method. The returned value is the new DOM node wrapped in a new lmx instance. Text nodes can be added to an lmx instance by calling the special method `text`.

lmx can enable you to quickly generate a large amount of HTML programatically. For example, to build a calendar for the current month, create a Widget which uses lmx to add DOM nodes to the incoming template node. Here is a complete example which when placed in an rpy and visited through the web will render the current month's calendar:

```
import time
import calendar
calendar.setfirstweekday(calendar.SUNDAY)

from twisted.web.microdom import lmx
from twisted.web.woven import widgets

class Calendar(widgets.Widget):
    def setUp(self, request, node, data):
```

```
        node.tagName = "table"
        curTime = time.localtime()
        curMonth = calendar.monthcalendar(curTime[0], curTime[1])
        today = curTime[2]
        month = lmx(node)
        headers = month.tr()
        for dayName in ["Su", "M", "T", "W", "Th", "F" , "S"]:
            headers.td(
                _class="dayName", align="middle"
            ).text(dayName)
        for curWeek in curMonth:
            week = month.tr(_class="week")
            for curDay in curWeek:
                if curDay == 0:
                    week.td(_class="blankDay")
                else:
                    if curDay == today:
                        className = "today"
                    else:
                        className = "day"
                    week.td(
                        _class=className, align="right"
                    ).text(str(curDay))


from twisted.web.woven import page


resource = page.Page(template="""<html>
    <head>
        <style type="text/css">
.week {  }
.day { border: 1px solid black; height: 2em; width: 2em; color: blue }
.today { border: 1px solid red; height: 2em; width: 2em; color: red }
.blankDay { height: 2em; width: 2em;}
        </style>
    </head>
    <body>
        <div view="calendar" />
    </body>
</html>""")

resource.setSubviewFactory("calendar", Calendar)
```

8.4.6 wvupdate_

Sometimes, you need to create some view-rendering code for a very specific purpose. Since this code will most likely not be reusable across pages, it is irritating to have to create a Widget just to house this code. Thus, Woven allows you to place specially named `wvupdate_` methods on your Page subclass. Think of a `wvupdate_` method as a setUp method that lives on the Page class. When Woven encounters a view= directive that matches with a `wvupdate_` method name, it will create a generic Widget instance and call the `wvupdate_` method instead of `setUp`.

Note that the generic Widget instance is passed in as the third argument to a `wvupdate_` method instead of a DOM node instance. Often this fact is not important, however, if you wish to access a Widget API such as getPattern, you must do so using the widget argument rather than self:

```
class MyPage(page.Page):
    def wvupdate_foo(self, request, widget, data):
        if data > 10:
            newNode = widget.getPattern("large")
        else:
            newNode = widget.getPattern("small")
        newNode.appendChild(request.d.createTextNode(str(data)))
        widget.appendChild(newNode)
```

It is often possible to use `wvupdate_` methods to quickly prototype some View code, and generalize this code later. By moving the `wvupdate_` code into a Widget subclass, you make this code available to many different Pages.

8.4.7 The View stack

Woven uses a View stack to keep track of which View objects are currently in scope. You can use this fact to provide View objects which contain a lot of view-manipulation logic, but are still cleanly implemented. When Woven encounters a node with a `view=` directive, it locates the View (by looking for a `wvfactory_` method) and places it on the View stack.

While this node is being rendered, the new View is in scope, and is searched for `wvfactory_` methods before other Views and the Page object. Thus you can create a View which is made up of other smaller parts:

```
from twisted.web.woven import view, page

class ShowHide(view.View):
    template = """<span>
    <div view="hider">
        <div pattern="contents" view="contents">
            Here are the contents!
        </div>
    </div>
</span>"""

    def wvupdate_hider(self, request, widget, data):
        ## We want the widget to be cleared before rendering
        widget.clearNode = 1
```

```
        hidden = int(request.args.get("hide", [0])[0])

        if hidden:
            opener = request.d.createElement("a")
            opener.setAttribute("href", "?hide=0")
            opener.appendChild(request.d.createTextNode("Open"))
            widget.appendChild(opener)
        else:
            closer = request.d.createElement("a")
            closer.setAttribute("href", "?hide=1")
            closer.appendChild(request.d.createTextNode("Close"))
            widget.appendChild(closer)
            widget.appendChild(widget.getPattern("contents"))

    def wvupdate_contents(self, request, widget, data):
        widget.clearNode = 1
        widget.appendChild(request.d.createTextNode("Some contents here"))

resource = page.Page(template="""<html>
    <body>
        <span view="showHide" />
    </body>
</html>""")

resource.setSubviewFactory("showHide", ShowHide)
```

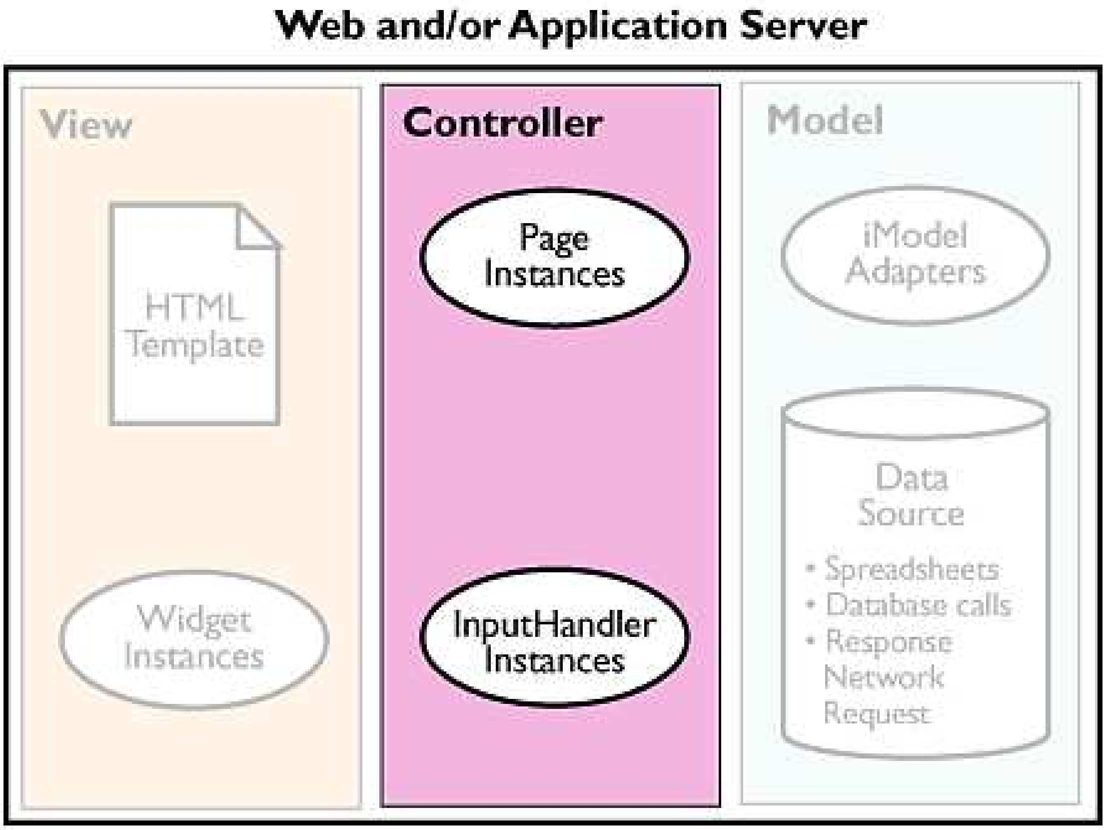
8.5 Controllers in Depth

Note:

This HOWTO documents the Controller objects, part of the Woven framework. The Woven framework should not be used for new projects. The newer Nevow[10] framework, available as part of the Quotient[11] project, is a simpler framework with consistent semantics and better testing and is strongly recommended over Woven.

The Woven documentation below is maintained only for users with an existing Woven codebase.

[10] http://www.divmod.org/Home/Projects/Nevow/
[11] http://www.divmod.org/Quotient/



Controller objects are a way to generalize and reuse input handling logic. In Twisted Web, form input is passed to a Resource instance in request.args. You can create controller classes to encapsulate generic request.args handling, and perform validation and Model updating tasks.

8.5.1 Main Concepts

- *Controller factories* (page 298) provide the glue from a DOM node with a 'controller=' directive to an instance of a Controller class.
- *handle* (page 298) is the method which is called on the Controller instance to handle a node.
- *InputHandlers* (page 298) are Controllers which have (somewhat) convenient syntax for handling a node.
- *Event handlers* (page 299), when used with *LivePage* (page 300), are a brain-exploding way of handling JavaScript events in your pages with server-side Python code.

8.5.2 Controller factories

Controller factories provide the glue between a `controller=` directive on a DOM node and a Controller instance. When a DOM node with a `controller=` directive is encountered, Woven looks for a corresponding `wcfactory_` method on your Page instance. A Controller factory is required to return an object which implements the interface `IController`.

```
class MyController(controller.Controller):
    pass

class MyPage(page.Page):
  def wcfactory_foo(self, request, node, model):
    return MyController(model)
```

8.5.3 Handle

Handle is the API your controller must implement to handle a node. It's return value may be a Deferred if you wish to pause the rendering of the View until some data is ready, or it may be None

```
class MyController(controller.Controller):
  def handle(self, request, node):
    name = request.args.get("name", [None])[0]
    print "HOORJ! YOUR NAME IS %s" % name
```

8.5.4 InputHandlers

InputHandlers are defined in `woven.input`. They were an early attempt to create a class which made it easy to create new input validators and input committers. It is usable in its current state, although the API is a bit baroque. Subclasses of `input.InputHandler` can override the following methods to decide what to do with data

`initialize()` initialize this Controller. This is most useful for registering event handlers on the View with *addEventHandler* (page 299), discussed below.

`getInput(self, request)` get input from the request and return it. Return None to indicate no data was available for this InputHandler to handle.

`check(self, request, data)` Check the input returned from getInput and return:

- None if no data was submitted (data was None), or
- True if the data that was submitted was valid, or
- False if the data that was submitted was not valid.

`handleValid(self, request, data)` handle the valid submission of some data. By default this calls `self.parent.aggregateValid`.

`aggregateValid(self, request, inputhandler, data)` Some input was validated by a child Controller. This is generally implemented on a controller which is placed on a `<form>` to gather input from controllers placed on `<input>` nodes.

`handleInvalid(self, request, data)` handle the invalid submission of some data. By default this calls `self.parent.aggregateInvalid`.

`aggregateInvalid(self, request, inputhandler, data)` Some input was declared invalid by a child Controller. This is generally implemented on a controller which is placed on a `<form>` to gather input from controllers placed on `<input>` nodes.

`commit(self, request, node, data)` Enough valid input was gathered to allow us to change the Model.

InputHandlers have been parameterized enough so you may simply use a generic InputHandler rather than subclassing and overriding:

```
class MyPage(page.Page):
  def checkName(self, request, name):
    if name is None: return None
    # No fred allowed
    if name == 'fred':
      return False
    return True

  def commitName(self, request, name=""):
    ctx = getContext()
    ctx.execute("insert into people (name) values %s", name)

  def wcfactory_addPerson(self, request, node, model):
      return input.InputHandler(
          model,
          name="name", # The name of the argument in the request to check
          check=self.checkName,
          commit=self.commitName)
```

8.5.5 Event handlers

Note:
In order for Event Handlers to work, you must be using *LivePage* (page 300), and include the webConduitGlue View in your HTML template.

Event handlers give you the powerful ability to respond to in-browser JavaScript event handlers with server-side Python code. Event handlers are registered on the View instance; in some cases, it may make most sense for your View instances to implement their own event handlers. However, in order to support good separation of concerns and code reuse, you may want to consider implementing your event handlers on a Controller instance.

The easiest way to achieve this is to subclass `input.Anything` (XXX: this should just be controller.Controller) and override `initialize` (XXX: this should be setUp):

```
class MyEventHandler(input.Anything):
  def initialize(self):
      self.view.addEventHandler("onclick", self.onClick)
      self.view.addEventHandler("onmouseover", self.onMouseOver, "'HELLO'")
```

```
    def onClick(self, request, widget):
        print self, "CLICKED!!!"

    def onMouseOver(self, request, widget, argument):
        print self, "MOUSE OVER!!!", argument
```

Note that the first argument to addEventHandler is the JavaScript event name, and the second argument is the python function or method which will handle this event. You may also pass any additional arguments you desire. These arguments must be valid JavaScript, and will be evaluated in the browser context. The results of these JavaScript expressions will be passed to your Python event handler.

Note that when we passed an extra argument when adding an `onmouseover` event handler, we passed a string enclosed in two sets of quotes. This is because the result of evaluating `"'HELLO'"` as JavaScript in the browser is the string `'HELLO'`, which is then passed to the Python event handler. If we had simply passed `"HELLO"` to addEventHandler, Woven would have evaluated `"HELLO"` in the browser context, resulting in an error because the variable `HELLO` is not defined.

Any normal client-side JavaScript object may be accessed, such as `document` and `window`. Also, the JavaScript variable `node` is defined as the DOM node on which the event handler is operating. This is useful for examining the current value of an `<input>` node.

Here are some examples of useful Event handlers:

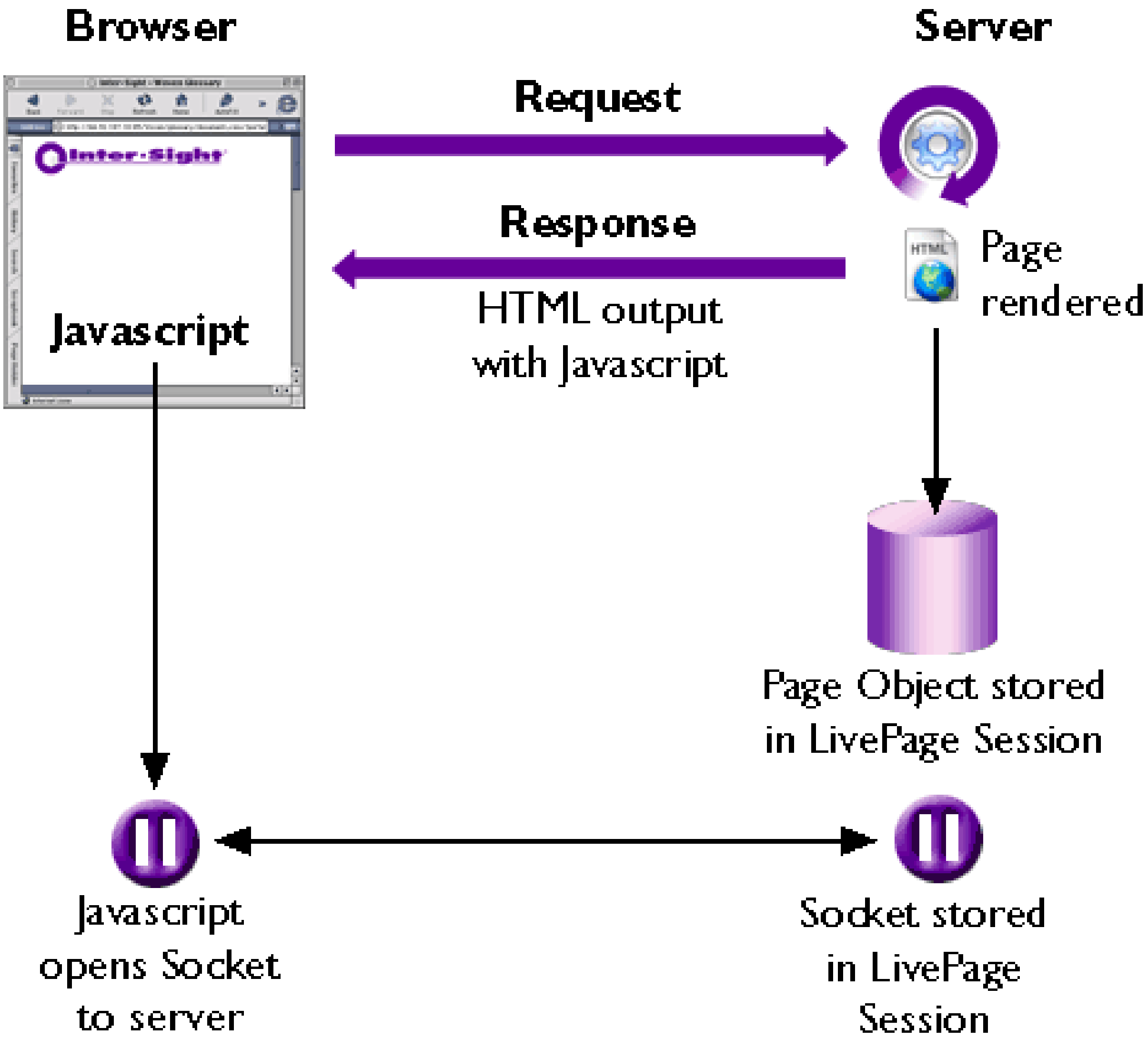
```
  class Redirect(input.Anything):
    def initialize(self):
        self.view.addEventHandler(
            "onclick",
            self.onClick,
            "window.location = 'http://www.google.com'")

    def onClick(self, request, widget, arg):
        print "The window was redirected."

  class OnChanger(input.Anything):
    def initialize(self):
        self.view.addEventHandler(
            "onchange",
            self.changed,
            "node.value")

    def changed(self, request, widget, newValue):
        print "The input box changed to", newValue
```

8.6 LivePage

Note:
This HOWTO documents the LivePage class, part of the Woven framework. The Woven framework

should not be used for new projects. The newer Nevow[12] framework, available as part of the Quotient[13] project, is a simpler framework with consistent semantics and better testing and is strongly recommended over Woven.

The Woven documentation below is maintained only for users with an existing Woven codebase. The LivePage HOWTO is just a quick bootstrap page in any case.



To use LivePage, subclass or instantiate `LivePage` instead of `Page`. Then, in your HTML template, include the following HTML fragment somewhere towards the bottom of the page:

[12] http://www.divmod.org/Home/Projects/Nevow/
[13] http://www.divmod.org/Quotient/

```
<span view="webConduitGlue" />
```

Then, *Event handlers* (page 299) can forward client-side JavaScript events to the Server, and you can send JavaScript to the browser after a page has already loaded. Example:

```
class Foo(page.LivePage):
    template = ('<html><body>'
                'Nothing here!<span view="webConduitGlue" />'
'</body></html>')
    def setUp(self, request):
        global currentPage

        # don't save this in a global but somewhere your code can get
        # to it later
        currentPage = request.getSession(interfaces.IWovenLivePage)

# then later, in response to some server-side event
def fooHappened():
    global currentPage

    currentPage.sendScript("alert('hello, world!')")
```

8.7 Page In Depth

Note:

This HOWTO documents the Page class, part of the Woven framework. The Woven framework should not be used for new projects. The newer Nevow[14] framework, available as part of the Quotient[15] project, is a simpler framework with consistent semantics and better testing and is strongly recommended over Woven.

The Woven documentation below is maintained only for users with an existing Woven codebase.

Page objects are the glue between a web request, a Model object, and a Template.

8.7.1 Main Concepts

- *Root Models* (page 303) are passed to Page objects when they are constructed. If no Model is passed, a Page will act as its own Root Model.
- *Templates* (page 303) for Page objects can be specified in various ways.
- *Child Pages* (page 304) for handling the next URL segment can be constructed using a convenient syntax supported by Page.
- *Factories* (page 305) for View and Controller objects are contained in the Page. The View factories and Controller factories will be matched up with `view=` and `controller=` directives located in the template during the rendering process.

[14] http://www.divmod.org/Home/Projects/Nevow/
[15] http://www.divmod.org/Quotient/

8.7.2 Root Models

A Root Model is the base of a Woven Model tree. All Model data that the Page will use for rendering should be made available through this Model. Any Python object can be used as a Model within Woven, even Dictionaries, Lists, Strings, Integers, and Floats. This is accomplished through the use of IModel adapters, which normalize Model access methods.

A Root Model is passed to the Page constructor as the first argument:

```
model = {'name': 'Donovan',
               'interests': ['Computers', 'Music', 'Bling Bling']
               }

resource = page.Page(model)
```

If no Model object is passed as the first argument to the Page constructor, the Page object itself will be treated like a MethodModel. See *Models* (page 283) for more information about Model objects.

Using a Page object as a Model factory (by not passing a Root Model) is discussed below in *Factories* (page 305).

8.7.3 Templates

The Template a page will render can be specified in various ways.

- Passing template="<html>Hello</html>"
- Passing templateFile="Template.xhtml"
- Passing both templateFile and templateDirectory="/Users/dsp/Templates"

Every Page object must be able to find a Template in order to render itself. There are three main ways a template can be specified. The first is simply by passing the template as a keyword argument to the Page constructor:

```
resource = page.Page(template="<html>Hello world!</html>")
```

However, it is desireable to store templates as separate HTML files on disk, where they can be edited easily by an external HTML editor. This can be accomplished by placing the template next to the .rpy script in the twisted.web directory and passing templateFile to the Page constructor:

```
resource = page.Page(templateFile="MyTemplate.xhtml")
```

Finally, you may wish to place templates in a specific location, away from the python code entirely. To do so, pass both templateFile and templateDirectory to the Page constructor, indicating where you would like the template to be found:

```
resource = page.Page(
  templateFile="MyTemplate.xhtml",
  templateDirectory="/Users/dsp/Templates")
```

If you are subclassing Page to provide child, model, view, or controller factories (discussed below), you may wish instead to specify a Page's template using a class attribute:

```
class MyPage(page.Page):
    templateFile = "MyTemplate.xhtml"
```

A useful technique for storing your templates in a place where they are conveniently located next to your Python modules is to define templateDirectory as a class attribute, using Python's introspection abilities to discover where your Python module is located:

```
class MyPage(page.Page):
    templateFile = "MyTemplate.xhtml"
    templateDirectory = os.path.join(os.path.split(__file__)[0], "templates")
```

How you manage your templates is up to you.

8.7.4 Child Pages

As discussed in the *Twisted Web* (page 247) section of the documentation, Resource objects provide access to their children by providing a getChild method which returns the next Resource object in the URL tree. Woven Page instances implement IResource and thus follow the same rules as any other Resource instance.

Woven Page instances can be built into Resource Trees in the same manner as regular Resources, using putChild. However, it is often convenient to construct a Page instance on the fly with a method.

Page instances with methods prefixed with `wchild_` will be invoked in getChild if there is a method matching the requested name. For example, if an instance of this class is used as the root Resource object, several URLs will be valid:

```
class MyPage(page.Page):
    template = """<html>
        Root Page
        <p><a href="fred">Fred</a></p>
        <p><a href="bob">Bob</a></p>
    </html>"""

    def wchild_fred(self, request):
        return page.Page(template="<html>Fred!</html>")

    def wchild_bob(self, request):
        return page.Page(template="<html>Bob!</html>")
```

The following URLs would then be valid:

- `http://example.com/`
- `http://example.com/fred`
- `http://example.com/bob`
- `http://example.com/fred/`
- `http://example.com/bob/`

There is one special wchild method for when the URL ends in a slash. When `twisted.web` calls `getChild` with an empty string (when the URL ends in a slash), the wchild method that is called is `wchild_index`.

By default, all Page instances will attempt to add a slash to the end of the URL if they are visited through the web. Thus, visiting `http://example.com/fred` in the above example will redirect your browser to `http://example.com/fred/`. This generally makes writing relative links to other pages easier.

If this is not the behavior you would like, define the class attribute `addSlash` to be false:

```
class MyPage(page.Page):
    addSlash = False
    template = "<html>No slash for you!</html>"
```

8.7.5 Factories

So far, we have observed the use of the special `model=`, `view=`, and `controller=` attributes ("directives") in Woven templates, but have not discussed how these attributes are translated to Python code.

There are three types of Woven factories:

1. *View factories* (this page)
2. *Controller factories* (page 306)
3. *Model factories* (page 306)

View factories

When a view directive is encountered in a Woven template, it is translated into a `wvfactory_` call. For example, the node `<div view="cool" />` will cause the method `wvfactory_cool` to be called on your Page instance. View factories are methods which have the following signature, and must return an IView implementor:

```
    def wvfactory_cool(self, request, node, model):
        return widgets.Text(model)
```

Widget subclasses are the most common return value from `wvfactory_` methods. Widgets are discussed in the *Views* (page 289) section. However, Page is also an implementor of IView, and you may take advantage of this to insert HTML fragments from other locations into a page which is being rendered:

```
class MyPage(page.Page):
    template = """<html>
    <body>
        Some stuff goes here.
        <div view="header" />
        Some more stuff goes here.
        <div view="body" />
        Even more stuff goes here.
    </body>
</html>"""

    def wvfactory_header(self, request, node, model):
        return page.Page(template="<div>This is the header.</div>")

    def wvfactory_body(self, request, node, model):
        return page.Page(template="<div>This is the body.</div>")
```

There is also a convenient special method, wvupdate_, which reduces the amount of boilerplate code required to quickly modify the template. wvupdate_ methods have a slightly different signature; instead of being passed the DOM node which contained the view= directive, they are passed an instance of the generic Widget class, and they are passed the unwrapped Model data rather than the IModel wrapper:

```
def wvupdate_red(self, request, widget, data):
    widget.setAttribute('style', 'color: red')
```

See the *Views* (page 289) section for more information on writing your application's View code.

Note that if an appropriately named View factory is not found on your page class, Woven will look for the name in the widgets module before giving up and raising an exception. This is why we have been able to create templates that have nodes like <div view="Text" /> and <div view="List" /> without getting exceptions.

Controller factories

When a controller directive is encountered in a Woven template, it is translated into a wcfactory_ call. For example, the node <input type="text" name="foo" controller="number" /< will cause the method wcfactory_number to be called on your Page instance. Controller factories are methods which have the following signature, and must return an IController implementor:

```
def wcfactory_number(self, request, node, model):
    return input.Integer(model)
```

The IController implementation classes which validate input currently live in the input module. This will probably be changed in the future.

See the *Controllers* (page 296) section for more information on writing your application's Controller code.

Note that if an appropriately named Controller factory is not found on your page class, Woven will look for the name in the input module before giving up and raising an exception.

Model factories

If your Page instance is passed a Root Model composed of basic python types, Woven is able to use IModel adapters to allow your Template to access the entire Model tree automatically. However, often it can be useful to produce your model data in some sort of method call which retrieves the data.

If your Page instance was initialized without a Root Model object, Woven will use the Page instance itself as a MethodModel. When a model directive is encountered in a Woven template, it will be translated into a wmfactory_ call on your Page instance. For example, the node <div model="name" /> will cause the method wmfactory_name to be called. Model factories are methods which have the following signature, and may return any Python object:

```
def wmfactory_name(self, request):
    return "Fred Bob"
```

See the *Models* (page 283) section for more information on writing your application's IModel adapters.

8.8 Form In Depth

XXX: To be written

8.9 Guard In Depth

Note:
This HOWTO documents Woven Guard, part of the Woven framework. The Woven framework should not be used for new projects. The newer Nevow[16] framework, available as part of the Quotient[17] project, is a simpler framework with consistent semantics and better testing and is strongly recommended over Woven.

Woven guard is founded on the *twisted.cred* (page 149) framework, and it is important to be somewhat familiar with cred to work with guard. As always, when using this framework, start by writing a `IRealm`. The guard module expects the realm to respond to the `resource.IResource` interace.

Here is an example of a simple realm which generates different resources for logged-in and non-logged-in users:

```
from twisted.cred import portal, checkers
from twisted.web import resource, static

def noLogout():
    pass

class Realm:
    __implements__ = portal.IRealm
    def requestAvatar(self, avatarId, mind, *interfaces):
        if resource.IResource not in interfaces:
            raise NotImplementedError
        if avatarId is checkers.ANONYMOUS:
            avatar = static.Data("Who are you?", 'text/html')
        else:
            avatar = static.Data("The answer is 42", 'text/html')
        return (resource.IResource, avatar, noLogout)
```

This realm makes sure that only non-anonymous users will know the answer to life, the universe and everything.

Usually, one or more custom resource classes will be written when using guard, but it is not necessary. For example, a guard intended to protect a certain directory can use a `static.File` resource. Note that the `IRealm` should be prepared to get the `ANONYMOUS` avatar ID, and handle it correctly. From whatever resource the anonymous avatar gets, a link should point to `guard.INIT_PERSPECTIVE` relative to the guard root. This is the only way users can log in. From whatever resources the non-anonymous avatars get it is strongly recommended to link to `guard.DESTROY_PERSPECTIVE`. This is the only way users can log out.

```
from twisted.cred import portal, checkers
from twisted.web import resource, static
from twisted.web.woven import guard

login='<a href="%s">login</a>' % guard.PERSPECTIVE_INIT
logout='<a href="%s">logout</a>' % guard.PERSPECTIVE_DESTROY
```

[16]http://www.divmod.org/Home/Projects/Nevow/
[17]http://www.divmod.org/Quotient/

```
def noLogout():
    pass

class Realm:
    __implements__ = portal.IRealm
    def requestAvatar(self, avatarId, mind, *interfaces):
        if resource.IResource not in interfaces:
            raise NotImplementedError
        if avatarId is checkers.ANONYMOUS:
            avatar = static.Data("Who are you?<br>"+login, 'text/html')
        else:
            avatar = static.Data("The answer is 42, %s<br>%s" % (avatarId,
                                 logout), 'text/html')
        return (resource.IResource, avatar, noLogout)
```

Once the realm is written, it is possible to generate a resource which will wrap it with appropriate code to manage users, sessions and authentication. But, as always, nothing deals with the realm directly – all the rest of the code deals with a `Portal` which wraps the realm.

> **Note:** You will almost always want to put `checkers.AllowAnonymousAccess` in the checkers registered for the `Portal`, otherwise it will be impossible to log in.

The canonical thing is to use `resource=guard.SessionWrapper(guard.UsernamePassword Wrapper(portal, callback=callback)`. The `callback` is used to redirect the request to the appropriate place after a successful login. Usually, you will want to redirect to the parent:

```
def parentRedirect(_):
    return util.ParentRedirect()
```

When a client first reaches a guarded resource, it is redirected to `session-init/`. From there, it will be redirected to a URL containing a long magic hex string, where a cookie will be set, and then to the original URL with `?__session_just_started__=1` tucked at the end. The addition is to guarantee that the client will not think it is in a redirection loop (wget, for example, has that problem).

Note that in resources which are children of the guarded resources, `request.getSession` automatically returns the Woven session. Since it is a `Componentized`, it is possible to use `getComponent` and `setComponent` to keep state related to a user in the session.

For simple cases, the approach described here leads to quite a bit of boiler-plate code (about 30 lines or so). If a web application has simple authentication needs, it is possible to use `simpleguard`, which allows you to skip implementing a realm yourself.

The important function in `simpleguard` is `resource=guardResource(resource, checkers, nonauthenticated=None)`. `checkers` should be a list of `ICredentialCheckers`. It is not necessary to put `AllowAnonymousAccess` here – it will be added automatically. This allow you to differentiate in resources only based on authenticated/anonymous users, without finer distinction. However, in the given resources, and their children, it is possible to use `request.getComponent(simpleguard.Authenticated)`. This will return `None` if the request is anonymous, or an instance with a `.name` attribute, which is the avatar ID. This can allow the application to personalize parts of it.

The way the login page and error-on-login page look can be customized when creating the guarded resource. Here is an example:

```
from twisted.web.woven import simpleguard, page, guard
from twisted.web import resource, util, microdom
from twisted.cred import checkers, portal

class Authenticated(page.Page):

    template='<html>Hello <span model="name"/>!</html>'

    def wmfactory_name(self, request):
        return request.getComponent(simpleguard.Authenticated).name

class LoginPage(page.Page):
    """This is the page that is shown to non-logged in users."""

    isLeaf = True
    addSlash = 0
    template = '''<html>
    <head>
        <title>Login</title>
        <style type="text/css">
.formDescription, .formError {
    /* fixme - inherit */
    font-size: smaller;
    font-family: sans-serif;
    margin-bottom: 1em;
}

.formDescription {
    color: green;
}

.formError {
    color: red;
}
</style>
    </head>
    <body>
    <h1>Please Log In</h1>
    <div class="shell">
    <div class="loginform" view="loginform" />
    </div>

    </body>
</html>'''

    def __init__(self, formModel=None):
```

```
        page.Page.__init__(self)
        self.formModel = formModel

    def wvupdate_loginform(self, request, widget, model):
        microdom.lmx(widget.node).form(action=guard.INIT_PERSPECTIVE,
                                       model="form")

    def wmfactory_form(self, request):
        if self.formModel:
            return self.formModel
        else:
            return guard.newLoginSignature.method(None)


def callback(_):
    return util.Redirect(".")

def buildGuardedResource():
    return simpleguard.guardResource(
               Authenticated(),
               [checkers.InMemoryUsernamePasswordDatabaseDontUse(bob="12345")],
               nonauthenticated=LoginPage(),
               callback=callback, errback=LoginPage)
```

listings/guard/login.py — *login.py*

The trick here is that, of course, we define the *original* login page: while `guard.PERSPECTIVE_INIT` gives a default form, it is quite all right to use a different one. In this example, we just use the home page of non-authenticated users to show the login form. There is one other case where the default form will be shown: if the attempt to login did not succeed. However, this is only what happens by default: we can override this by supplying an `errback`. This will be called with a model describing the form, and all we have to do is to use woven to display it.

Note that this example uses `simpleguard`, but equivalent code can be developed using `guard` – the errback should be passed to the `UsernamePasswordWrapper` constructor in that case.

8.10 Woven Widget Reference

Note:

This document is a reference guide for Woven builtins and signatures. However, the Woven framework should not be used for new projects. The newer Nevow[18] framework, available as part of the Quotient[19] project, is a simpler framework with consistent semantics and better testing and is strongly recommended over Woven.

The Woven documentation below is maintained only for users with an existing Woven codebase.

[18]http://www.divmod.org/Home/Projects/Nevow/
[19]http://www.divmod.org/Quotient/

If you have feedback on this guide, please send it[20]!

8.10.1 Common Factory Signatures

```
class MyPage(page.Page):
    def wchild_foo(self, request):
        """ Return the Resource for /mypage/foo/ """

    def wmfactory_foo(self, request):
        """ Return the Model object for model="/foo" """

    def wvfactory_cool(self, request, node, model):
        """ Return node for view="cool" """

    def wvupdate_thumbnail(self, request, widget, data):
        """ Update and Return the widget for view="thumbnail" """

    def wcfactory_adjuster(self, request, node, model):
        """ Return the controller for controller="adjuster" """
```

8.10.2 Built-in Widgets

Attributes

For dressing a node with extra attributes from the model.

```
class MyPage(page.Page):
    template = """
    <a view="Attributes" model="/mylink"> My Link! </a>
    """

MyPage({'mylink': {'href': '/path/to'}})
```

Text, RawText

For putting text into a node. This will escape any HTML/XML special characters turning them into HTML entities, i.e. <foo>.

```
class MyPage(page.Page):
    template = """
    <span view="Text" model="/mytext" />
    """

MyPage({'mytext': 'abc'})
```

[20] mailto:stephen@thorne.id.au

ParagraphText

For taking text, one paragraph per line, and dressing it with <p> tags.

```
class MyPage(page.Page):
    template = """
    <span view="ParagraphText" model="/mytext" />
    """

MyPage({'mytext' : """
This is one Paragraph.
This is a second one. It has two sentences.
"""})
```

Image

Displaying Images. Easy.

```
class MyPage(page.Page):
    template = """
    <img view="Image" alt="MyImage" model="/myimg" />
    """

MyPage({'myimg': '/images/myimage.png'})
```

Error

Displaying an error. In red.

Div

A Div.

Span

A Span.

Br

A Br.

Input

An Input, it takes `value`, `name` and `id`. If `name` isn't supplied, it reverts to the value of `id`, if `id` isn't supplied either, it takes the name of the submodel. The other input classes extend this one, and give it a proper `type`.

CheckBox

As Input, but... boxy.

RadioButton

As Input, but... kinda button like.

File

As Input, but... more chance of 1's and 0's. And a dialog on some user-agents.

Password

Renders a password text field. no option for maxlength, quite strange.

Button

Button

Select

This falls under input, but instead of taking a value, you have to fiddle with options for that. There's no way to specify `type="multiple"` either it seems.

Option

Option widgets have the methods `setText` and `setValue`, but if your text and your value co-incide, they can just be put in the model and it will all be hunky and/or dory.

Anchor

To create an anchor, including safe handling of parameters.

```
class MyPage(page.Page):
    template = """
    <a view="Anchor" model="/anchorone" />
    <a view="myanchor" model="/anchortwo" />
    """

    def wvfactory_myanchor(self, request, node, model):
        a = Anchor()
        a.setText(model['text'])
        a.setParameter('thingy', model['thingy'])
        a.setLink(model['href'])

MyPage({
    'anchorone': '/pages/otherpage',
    'anchortwo': {
        'href': '/pages/thisotherpage/',
        'thingy': 'foobar',
        'text': 'The Text Of The Link'
```

```
    }
})
```

DirectoryAnchor

Exactly like Anchor, but given a model which contains `/pages/otherpage` it will provide a link to `/pages/otherpage/` with a trailing slash.

List

List is a nice view for displaying the contents of a list. You can choose not to specifiy listHeader, listFooter and emptyList. Specifying listItem twice or more will cause alternation.

```
class MyPage(page.Page):
    template = """
         <table model="blah" view="List">
             <tr pattern="listHeader"><th>A</th><th>B</th></tr>
             <tr pattern="emptyList"><td colspan='2'>***None***</td></tr>
             <tr pattern="listItem">
                 <td><span view="Text" model="0" /></td>
                 <td><span view="Text" model="1" /></td>
             </tr>
             <tr pattern="listFooter"><td colspan="2">All done!</td></tr>
         </table>
    """

MyPage({'blah': [('abc', 'fza'), ('def', 'fdsa'), ('ghi', 'fdas')]})
```

KeyedList

KeyedList is similar to List above, works nearly the same way, except the model is a dictionary, not a list. If you need access to the key in your listItems, make your listItem use a view that will inspect the `model` attribute, as the KeyedList widget will create subwidgets, respecting `view` attribute but overriding `model`.

```
class MyPage(page.Page):
    template = """
         <table model="blah" view="List">
             <tr pattern="listHeader"><th>A</th><th>B</th></tr>
             <tr pattern="emptyList"><td colspan='2'>***None***</td></tr>
             <tr pattern="listItem">
                 <td><span view="Text" model="0" /></td>
                 <td><span view="Text" model="1" /></td>
             </tr>
             <tr pattern="listFooter"><td colspan="2">All done!</td></tr>
         </table>
    """

MyPage({'blah' : [('abc', 'fza'), ('def', 'fdsa'), ('ghi', 'fdas')]})
```

Bold

A Bold (b, shouldn't this make a 'strong'?).

Table

A Table.

Row

A Row (tr).

Cell

A Cell (td).

Link

A Link (a) tag. Similar to Anchor.

```
class MyPage(page.Page):
    template = """
    <a view="Link" model="/anchorone"> My Text </a>
    <a view="Link" model="/anchortwo" />
    """

MyPage({
    'anchorone': '/pages/otherpage',
    'anchortwo': {
        'href': '/pages/thisotherpage/',
        'text': 'The Text Of The Link'
    }
})
```

RootRelativeLink

Extends Link, and adds the functionality that it will respect this request.getRootURL, so that we can be deployed in different roots easily.

```
self['href'] = request.getRootURL() + '/' + self['href']
```

ExpandMacro

A Macro expansion widget modeled after the METAL expander in ZPT/TAL/METAL.

In the Page that is being rendered, place the ExpandMacro widget on the node you want replaced with the Macro, and provide nodes tagged with `fill-slot=` attributes which will fill slots in the Macro:

```
class MyPage(page.Page):
    def wvfactory_myMacro(self, request, node, model):
        return ExpandMacro(
            model,
            macroFile="MyMacro.xhtml",
            macroName="main")
```

```
<div view="myMacro">
    <span fill-slot="greeting">Hello</span>
    <span fill-slot="greetee">World</span>
</div>
```

Then, in your Macro template file ("MyMacro.html" in the above example) designate a node as the macro node, and nodes inside that as the slot nodes:

```
<div macro="main">
    <h3><span slot="greeting" />, <span slot="greetee" />!</h3>
</div>
```

DeferredWidget

Wraps around a deferred. Usage unknown.

Chapter 9

Dot Products

9.1 Creating and working with a names (DNS) server

A Names server can be perform three basic operations:

- act as a recursive server, forwarding queries to other servers
- perform local caching of recursively discovered records
- act as the authoritative server for a domain

Creating a non-authoritative server

The first two of these are easy, and you can create a server that performs them with the command `mktap dns --recursive --cache`, or launch `tkmktap` and configure a dns server with it. The result should be a file named `dns.tap`. Now switch to a superuser account (if required by your platform to bind to port 53) and run `twistd -f dns.tap`. The Application will run and bind to port 53. Try performing a lookup with it, `dig twistedmatrix.com @127.0.0.1`.

Creating an authoritative server

To act as the authority for a domain, two things are necessary: the address of the machine on which the domain name server will run must be registered as a nameserver for the domain; and the domain name server must be configured to act as the authority. The first requirement is beyond the scope of this howto and will not be covered.

To configure Names to act as the authority for `example-domain.com`, we first create a zone file for this domain.

```
zone = [
    SOA(
        # For whom we are the authority
        'example-domain.com',

        # This nameserver's name
        mname = "ns1.example-domain.com",
```

```
        # Mailbox of individual who handles this
        rname = "root.example-domain.com",

        # Unique serial identifying this SOA data
        serial = 2003010601,

        # Time interval before zone should be refreshed
        refresh = "1H",

        # Interval before failed refresh should be retried
        retry = "1H",

        # Upper limit on time interval before expiry
        expire = "1H",

        # Minimum TTL
        minimum = "1H"
    ),

    A('example-domain.com', '127.0.0.1'),
    NS('ns1.example-domain.com', 'example-domain.com'),

    CNAME('www.example-domain.com', 'example-domain.com'),
    CNAME('ftp.example-domain.com', 'example-domain.com'),

    MX('example-domain.com', 0, 'mail.example-domain.com'),
    A('mail.example-domain.com', '123.0.16.43')
]
```

Zone file — *example-domain.com*

Next, run the command `mktap dns --pyzone example-domain.com`, and then (as above) `twistd -f dns.tap`. Now try querying the domain locally (again, with dig): `dig -t any example-domain.com @127.0.0.1`.

Names can also read a traditional, BIND-syntax zone file. Specify these with the `--bindzone` parameter. The $GENERATE and $INCLUDE directives are not yet supported.

9.2 Using the Lore Documentation System

9.2.1 Writing Lore Documents

Overview

Lore documents are a special subset of XHTML documents. They use specific subset of XHTML, together with custom classes, to allow a wide variety of document elements, including some Python-specific ones. Lore documents, in particular, are well-formed XML documents. XML can be written using a wide variety of tools: from run of the mill editors such as vi, through editors with XML help like EMACS and ending with XML specific tools like (need name of XML editor here). Here, we will not cover the specifics of writing XML documents, except for a very broad overview.

XML documents contain elements, which are delimited by an opening tag which looks like <tag-name attribute="value"> and ends with a closing tag, which looks like </tag-name>. If an elements happen to contain nothing, it can be shortened to <tag-name />. Elements can contain other elements, or text. Text can contain any characters except <, > and &. These characters are rendered by <, > and &, respectively.

A Lore document is a single html element. Inside this element, there are exactly two top-level elements: head and body. The head element must contain exactly one element: title, containing the title of the document. Most of the document will be contained in the body element. The body element must start with an h1 (top-level header) element, which contains the exact same content as the title element.

Thus, a fairly minimal Lore document might look like:

```
<html>
<head><title>Title</title></head>
<body><h1>Title</h1></body>
</html>
```

Elements and Their Uses

p: The paragraph element. Most of the document should be inside paragraphs.

span: The span element is an element which has no meaning – unless it has a special class attributes. The following classes have the stated meanings:

footnote a small comment which should not be inside the main text-flow.

manhole-output This signifies, within a manhole transcript, that the enclosed text is the output and not something the user has to input.

span This should be an *empty* element, with an attribute value. That attribute should be an index term, in the format of generic!specific!more specific. Usually, you will only have one level, in which case value="term" works.

div: The div element is equivalent to a span, except it always appears outside paragraphs. The following classes have the given meanings:

note A short note which is not necessary for the understanding of the text.

doit An indication that the discussed feature is not complete or implemented yet.

boxed An indication that the text should be clearly separated from its surroundings.

a: This element can have several meanings, depending on the attributes:

name attribute Add a label to the current position, which might be used in this document or other documents to refer to.

href=URL Refer to some WWW resource.

`href=relative-path`, `href=relative-path#label` or `href=#label` Refer to a position in a Lore resource. By default, relative links to `.xhtml` files are changed to point to a `.html` file. If you need a link to a local non-Lore .xhtml file, use `class=absolute` to make Lore treat it as an absolute link.

`href=relative-path` with `class=py-listing` or `class=html-listing` Indicate the given resource is a part of the text flow, and should be inlined (and if possible, syntax highlighted).

`ol`, `ul`: A list. It can be enumerated or bulleted. Inside a list, the element `li` (for a list element) is valid.
`h2`, `h3`: Second- and third-level section headings.
`code`: a string which has meaning to the computer. There are many possible classes:

`API` A class, function or a module. It does not have to be a fully qualified name – but if it isn't, a `base` attribute is necessary. Example: `<code class="API" base="urllib">urlencode<code>`.

`shell` Shell (usually Bourne) code.

`python` Python code

`py-prototype` Function prototype

`py-filename` Python file

`py-src-string` Python string

`py-signature` Function signature

`py-src-parameter` Parameter

`py-src-identifier` Identifier

`py-src-keyword` Keyword

`pre`: Preformatted text, usually for file listings. It can be used with the `python` class to indicate Python syntax coloring. Other possible classes are `shell` (to indicate a shell-transcript) or `python-interpreter` (to indicate an interactive interpreter transcript).
`img`: Insert the image indicated by the `src` attribute.
`q`: The quote signs (`"`) are not recommended except in preformatted or code environment. Instead, quote by using the `q` element which allows nested quotes and properly distinguishes opening quote from closing quote.
`em`, `strong`: Emphasise (or strongly emphasise) text.
`table`: Tabular data. Inside a table, use the `tr` element for each rows, and inside it use either `td` for a regular table cell or `th` for a table header (column or row).
`blockquote`: A long quote which should be properly seperated from the main text.
`cite`: Cite a resource.
`sub`, `sup`: subscripts and superscripts.
`link`: currently, the only `link` elements supported are for for indicating authorship. `<link rel="author" href="author-address@examples.com" title="Author Name" />` should be used to indicate authorship. Multiple instances are allowed, and indicate shared authorship.

9.2.2 Writing Lore XHTML Templates

One of Lore's output formats is XHTML. Lore itself is very markup-light, but the output XHTML is much more markup intensive. Part of the auto-generated markup is directed by a special template.

The output of Lore is inserted into template in the following way:

- The title is appended into each element with class `title`.
- The body is inserted into the first element that has class `body`.
- The table of contents is inserted into the first element that has class `toc`.

In particular, most of the header is not tampered with – so it is easy to indicate a CSS stylesheet in the template.

9.2.3 Using Lore to Generate HTML

After having written a template, the easiest way to build HTML from the Lore document is by:

```
% lore --config template=mytemplate.tpl mydocument.xhtml
```

This will create a file called `mydocument.html`.

For example, to generate the HTML version of the Twisted docs from a CVS checkout, do:[1]

```
% lore --config template=doc/howto/template.tpl doc/howto/*.xhtml
```

In order to generate files with a different extension, use the `--config` commandline flag to tell the HTML output plugin to use a different extension:

```
% lore --config ext=.html doc/howto/*.xhtml
```

9.2.4 Using Lore to Generate LaTex

Articles

```
% lore --output latex mydocument.xhtml
```

Books

Have a Lore file for each section. Then, have a LaTeX file which inputs all the given LaTeX files. Generate all the LaTeX files by using

```
% lore --output latex --config section *.xhtml
```

in the relevant directory.

[1] See also the `admin/process-docs` script.

9.2.5 Using Lore to Generate Slides

Lore can also be used to generate slides for presentations. The start of a new slide is indicated by use of an h2 tag, with the content between the opening and closing tags the title of the slide. Slides are generated by

```
% lore --input lore-slides myslides.xhtml
```

This, by default, will produce HTML output with one HTML file for each slide. For our example, the files would be named myslides-<number>.html, where number is the slide number, starting with 0 for the title slide. Lore will look for a template file, either indicated by the `--config template=mytemplate.tpl` or the default template.tpl in the current directory. An example slide template is found in `doc/examples/slides-template.tpl`

The slides module currently supports three major output types: HTML, Magic Point, and LaTeX. The options for the latter two will be covered individually.

Magic Point Output

Lore supports outputting to the Magic Point file format. Magicpoint is a presentation program for X, which can be installed on Debian by `apt-get install mgp` or by visiting the Magic Point homepage[2] otherwise. A template file is required, `template.mgp` is shipped in the `twisted/lore` directory. Magic Point slides are generated by

```
% lore --input lore-slides --output mgp \
  --config template=~/Twisted/twisted/lore/template.mgp \
  myslides.xhtml
```

That will produce `myslides.mgp`.

LaTeX Output

Lore can also produce slides in LaTeX format. It supports three main styles: one slide per page, two per page, and Prosper format, with the `--config` parameters being `page`, `twopage`, and `prosper` respectively. Prosper is a LaTeX class for creating slides, which can be installed on Debian by `apt-get install prosper` or by visiting the Prosper homepage[3]. LaTeX format slides (using the Prosper option, for example) are generated by

```
% lore --input lore-slides --output latex \
  --config prosper myslides.xhtml
```

This will generate `myslides.tex` file that can be processed with `latex` or `pdftex` or the appropriate LaTeX processing command.

9.2.6 Linting

```
% lore --output lint mydocument.xhtml
```

This will generate compiler-style (file:line:column:message) warnings. It is possible to integrate these warnings into a smart editor such as EMACS, but it has not been done yet.

[2] http://www.mew.org/mgp/
[3] http://prosper.sourceforge.net/

9.3 Extending the Lore Documentation System

9.3.1 Overview

The *Lore Documentation System* (page 318), out of the box, is specialized for documenting Twisted. Its markup includes CSS classes for Python, HTML, filenames, and other Twisted-focused categories. But don't think this means Lore can't be used for other documentation tasks! Lore is designed to allow extensions, giving any Python programmer the ability to customize Lore for documenting almost anything.

There are several reasons why you would want to extend Lore. You may want to attach file formats Lore does not understand to your documentation. You may want to create callouts that have special meanings to the reader, to give a memorable appearance to text such as, "WARNING: This software was written by a frothing madman!" You may want to create color-coding for a different programming language, or you may find that Lore does not provide you with enough structure to mark your document up completely. All of these situations can be solved by creating an extension.

9.3.2 Inputs and Outputs

Lore works by reading the HTML source of your document, and producing whatever output the user specifies on the command line. If the HTML document is well-formed XML that meets a certain minimum standard, Lore will be able to to produce some output. All Lore extensions will be written to redefine the *input*, and most will redefine the output in some way. The name of the default input is "lore". When you write your extension, you will come up with a new name for your input, telling Lore what rules to use to process the file.

Lore can produce XHTML, LaTeX, and DocBook document formats, which can be displayed directly if you have a user agent capable of viewing them, or processed into a third form such as PostScript or PDF. Another output is called "lint", after the static-checking utility for C, and is used for the same reason: to statically check input files for problems. The "lint" output is just a stream of error messages, not a formatted document, but is important because it gives users the ability to validate their input before trying to process it. For the first example, the only output we will be concerned with is LaTeX.

Creating New Inputs

Create a new input to tell Lore that your document is marked up differently from a vanilla Lore document. This gives you the power to define a new tag class, for example:

```
<p>The Frabjulon <span class="productname">Limpet 2000</span>
is the <span class="marketinglie">industry-leading</span> aquatic
mollusc counter, bar none.</p>
```

The above HTML is an instance of a new input to Lore, which we will call MyHTML, to differentiate it from the "lore" input. We want it to have the following markup:

- A `productname` class for the <span> tag, which produces underlined text
- A `marketinglie` class for <span> tag, which produces larger type, bold text

Note that I chose class names that are valid Python identifiers. You will see why shortly. To get these two effects in Lore's HTML output, all we have to do is create a cascading stylesheet (CSS), and use it in the Lore XHTML Template. However, we also want these effects to work in LaTeX, and we want the output of lint to produce no warnings when it sees lines with these 2 classes. To make LaTeX and lint work, we start by creating a plugin.

```
register('MyHTML', "myhtml.factory", description="My Lore Plugin",
         type="lore", tapname="myhtml")
```

Listing 1: The Plugin File — *plugins.tml*

Name this file `plugins.tml`, and put it in a new package directory named `myhtml`. Also create an `__init__.py` file in your new package. Note that *the __init__ file can contain nothing but a doc string, but it must exist.* If `__init__.py` is empty, you will have problems with certain unzip programs that don't extract empty files.

The combination of plugin file and __init__ file causes the new package to be treated like a Twisted plugin, one that Lore knows how to make use of. The first three arguments to `register()` are the human-readable name, module name, and description of your plugin. The `type` parameter makes this plugin visible to the Lore system (rather than some other part of Twisted). The `tapname` parameter is an arbitrary filename with no extension; by convention you should use the lowercase version of your first argument (the human-readable name). Users of your extension will pass this argument to lore with the `--input` parameter on the command line. (For more details on plugins, read *Writing a New Plug-In for Twisted* (page 157).)

Let's look at that module name more closely: `myhtml.factory`. We will be creating this file next, in the package named `myhtml`. The purpose of the factory module is to tell Lore how to process your input.

```
from twisted.lore import default
import spitters

class MyProcessingFunctionFactory(default.ProcessingFunctionFactory):
    latexSpitters={None: spitters.MyLatexSpitter,
                  }

# initialize the global variable factory with an instance of your new factory
factory=MyProcessingFunctionFactory()
```

Listing 2: The Input Factory — *factory.py-1*

In Listing 2, we create a subclass of ProcessingFunctionFactory. This class provides a hook for you, a class variable named `latexSpitters`. This variable tells Lore what new class will be generating LaTeX from your input format. We redefine `latexSpitters` to `MyLatexSpitter` in the subclass because this class knows what to do with the new input we have already defined. Last, you must define the module-level variable `factory`. It should be an instance with the same interface as `ProcessingFunctionFactory` (e.g. an instance of a subclass, in this case, `MyProcessingFunctionFactory`).

Now let's actually write some code to generate the LaTeX. Doing this requires at least a familiarity with the LaTeX language. Search Google for "latex tutorial" and you will find any number of useful LaTeX resources.

```
from twisted.lore import latex
from twisted.lore.latex import processFile
import os.path

class MyLatexSpitter(latex.LatexSpitter):
    def visitNode_span_productname(self, node):
        # start an underline section in LaTeX
```

```
        self.writer('\\underline{')
        # process the node and its children
        self.visitNodeDefault(node)
        # end the underline block
        self.writer('}')

    def visitNode_span_marketinglie(self, node):
        # this example turns on more than one LaTeX effect at once
        self.writer('\\begin{bf}\\begin{Large}')
        self.visitNodeDefault(node)
        self.writer('\\end{Large}\\end{bf}')
```

Listing 3: spitters.py — *spitters.py-1*

The method `visitNode_span_productname` is our handler for <span> tags with the `class="productname"` identifier. Lore knows to try methods `visitNode_span_*` and `visitNode_div_*` whenever it encounters a new class in one of these tags. This is why the class names have to be valid Python identifiers.

Now let's see what Lore does with these new classes with the following input file:

```
<html>
  <head>
    <title>My First Example</title>
  </head>
  <body>
    <h1>My First Example</h1>
    <p>The Frabjulon <span class="productname">Limpet 2000</span>
    is the <span class="marketinglie">industry-leading</span> aquatic
    mollusc counter, bar none.</p>
  </body>
</html>
```

Listing 4: 1st_example.html — *1st_example.html*

First, verify that your package is laid out correctly. Your directory structure should look like this:

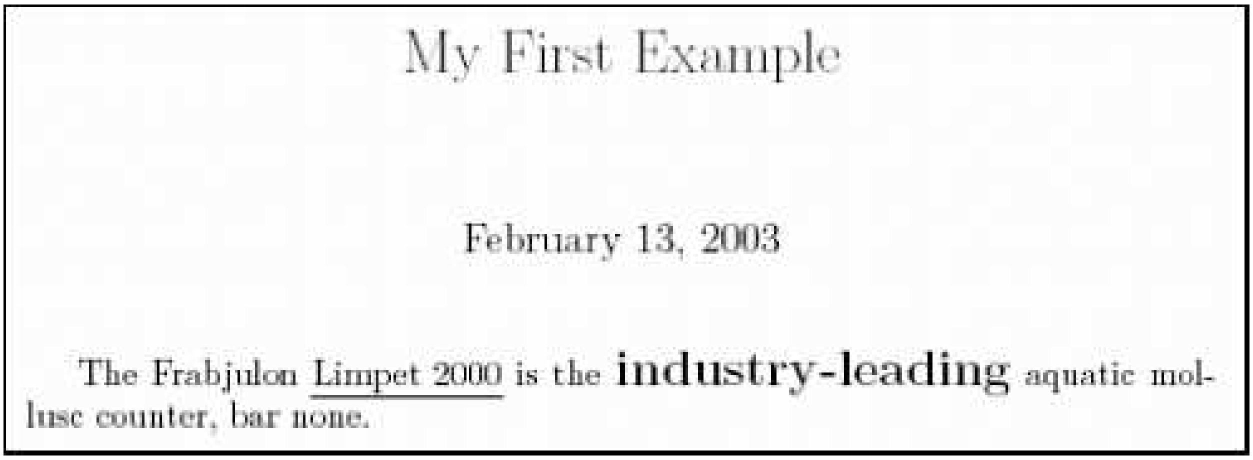
```
1st_example.html
myhtml/
       __init__.py
       factory.py
       plugins.tml
       spitters.py
```

In the parent directory of myhtml (that is, `myhtml/..`), run lore and pdflatex on the input:

```
$ lore --input myhtml --output latex 1st_example.html
[########################################] (*Done*)
```

```
$ pdflatex 1st_example.tex
[ . . . latex output omitted for brevity . . . ]
Output written on 1st_example.pdf (1 page, 22260 bytes).
Transcript written on 1st_example.log.
```

And here's what the rendered PDF looks like:



My First Example

February 13, 2003

The Frabjulon Limpet 2000 is the **industry-leading** aquatic mollusc counter, bar none.

What happens when we run lore on this file using the lint output?

```
$ lore --input myhtml --output lint 1st_example.html
1st_example.html:7:47: unknown class productname
1st_example.html:8:38: unknown class marketinglie
[########################################] (*Done*)
```

Lint reports these classes as errors, even though our spitter knows how to process them. To fix this problem, we must add to `factory.py`.

```
from twisted.lore import default
import spitters

class MyProcessingFunctionFactory(default.ProcessingFunctionFactory):
    latexSpitters={None: spitters.MyLatexSpitter,
                  }

    # redefine getLintChecker to validate our classes
    def getLintChecker(self):
        # use the default checker from parent
        checker = lint.getDefaultChecker()
        checker.allowedClasses = checker.allowedClasses.copy()
        oldSpan = checker.allowedClasses['span']
```

```
        checkfunc=lambda cl: oldSpan(cl) or cl in ['marketinglie',
                                                   'productname']
        checker.allowedClasses['span'] = checkfunc
        return checker

# initialize the global variable factory with an instance of your new factory
factory=MyProcessingFunctionFactory()
```

Listing 5: Input Factory with Lint Support — *factory.py-2*

The method `getLintChecker` is called by Lore to produce the lint output. This modification adds our classes to the list of classes lint ignores:

```
$ lore --input myhtml --output lint 1st_example.html
[#######################################] (*Done*)
$ # Hooray!
```

Finally, there are two other sub-outputs of LaTeX, for a total of three different ways that Lore can produce LaTeX: the default way, which produces as output an entire, self-contained LaTeX document; with `--config section` on the command line, which produces a LaTeX \section; and with `--config chapter`, which produces a LaTeX \chapter. To support these options as well, the solution is to make the new spitter class a mixin, and use it with the `SectionLatexSpitter` and `ChapterLatexSpitter`, respectively. Comments in the following listings tell you everything you need to know about making these simple changes:

- ```
  from twisted.lore import default
  import spitters

  class MyProcessingFunctionFactory(default.ProcessingFunctionFactory):
      # 1. add the keys "chapter" and "section" to latexSpitters to handle the
      # --config chapter and --config section options
      latexSpitters={None: spitters.MyLatexSpitter,
                     "section": spitters.MySectionLatexSpitter,
                     "chapter": spitters.MyChapterLatexSpitter,
                     }

      def getLintChecker(self):
          checker = lint.getDefaultChecker()
          checker.allowedClasses = checker.allowedClasses.copy()
          oldSpan = checker.allowedClasses['span']
          checkfunc=lambda cl: oldSpan(cl) or cl in ['marketinglie',
                                                     'productname']
          checker.allowedClasses['span'] = checkfunc
          return checker

  factory=MyProcessingFunctionFactory()
  ```

  factory.py — *factory.py-3*
  ```

- ```
  from twisted.lore import latex
  from twisted.lore.latex import processFile
  import os.path

  # 2. Create a new mixin that does what the old MyLatexSpitter used to do:
  # process the new classes we defined
  class MySpitterMixin:
      def visitNode_span_productname(self, node):
          self.writer('\\underline{')
          self.visitNodeDefault(node)
          self.writer('}')

      def visitNode_span_marketinglie(self, node):
          self.writer('\\begin{bf}\\begin{Large}')
          self.visitNodeDefault(node)
          self.writer('\\end{Large}\\end{bf}')

  # 3. inherit from the mixin class for each of the three sub-spitters
  class MyLatexSpitter(MySpitterMixin, latex.LatexSpitter):
      pass

  class MySectionLatexSpitter(MySpitterMixin, latex.SectionLatexSpitter):
      pass

  class MyChapterLatexSpitter(MySpitterMixin, latex.ChapterLatexSpitter):
      pass
  ```

spitters.py — *spitters.py-2*

Creating New Outputs

write some stuff

9.3.3 Other Uses for Lore Extensions

write some stuff

Color-Code Programming Languages

write some stuff

Add New Structural Elements

write some stuff
  ```

Support New File Formats

write some stuff

9.4 Writing a client with Twisted.Conch

9.4.1 Introduction

In the original days of computing, rsh/rlogin were used to connect to remote computers and execute commands. These commands had the problem that the passwords and commands were sent in the clear. To solve this problem, the SSH protocol was created. Twisted.Conch implements the second version of this protocol.

9.4.2 Writing a client

Writing a client with Conch involves sub-classing 4 classes: `twisted.conch.ssh.transport.SSHClientTransport`, `twisted.conch.ssh.userauth.SSHUserAuthClient`, `twisted.conch.ssh.connection.SSHConnection`, and `twisted.conch.ssh.channel.SSHChannel`. We'll start out with `SSHClientTransport` because it's the base of the client.

9.4.3 The Transport

```
from twisted.conch import error
from twisted.conch.ssh import transport
from twisted.internet import defer

class ClientTransport(transport.SSHClientTransport):

    def verifyHostKey(self, pubKey, fingerprint):
        if fingerprint != 'b1:94:6a:c9:24:92:d2:34:7c:62:35:b4:d2:61:11:84':
            return defer.fail(error.ConchError('bad key'))
        else:
            return defer.succeed(1)

    def connectionSecure(self):
        self.requestService(ClientUserAuth('user', ClientConnection()))
```

See how easy it is? `SSHClientTransport` handles the negotiation of encryption and the verification of keys for you. The one security element that you as a client writer need to implement is `verifyHostKey()`. This method is called with two strings: the public key sent by the server and its fingerprint. You should verify the host key the server sends, either by checking against a hard-coded value as in the example, or by asking the user. `verifyHostKey` returns a `twisted.internet.defer.Deferred` which gets a callback if the host key is valid, or an errback if it is not. Note that in the above, replace 'user' with the username you're attempting to ssh with, for instance a call to `os.getlogin()` for the current user.

The second method you need to implement is `connectionSecure()`. It is called when the encryption is set up and other services can be run. The example requests that the `ClientUserAuth` service be started. This service will be discussed next.

9.4.4 The Authorization Client

```
from twisted.conch.ssh import keys, userauth

# these are the public/private keys from test_conch

publicKey = 'ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAGEArzJx8OYOnJmzf4tfBEvLi8DVPrJ3\
/c9k2I/Az64fxjHf9imyRJbixtQhlH9lfNjUIx+4LmrJH5QNRsFporcHDKOTwTTYLh5KmRpslkYHR\
ivcJSkbh/C+BR3utDS555mV'

privateKey = """-----BEGIN RSA PRIVATE KEY-----
MIIByAIBAAJhAK8ycfDmDpyZs3+LXwRLy4vA1T6yd/3PZNiPwM+uH8Yx3/YpskSW
4sbUIZR/ZXzY1CMfuC5qyR+UDUbBaaK3Bwyjk8E02C4eSpkabJZGB0Yr3CUpG4fw
vgUd7rQ0ueeZlQIBIwJgbh+1VZfr7WftK5lu7MHtqE1S1vPWZQYE3+VUn8yJADyb
Z4fsZaCrzW9lkIqXkE3GIY+ojdhZhkO1gbG0118sIgphwSWKRxK0mvh6ERxKqIt1
xJEJO74EykXZV4oNJ8sjAjEA3J9r2ZghVhGN6V8DnQrTk24Td0E8hU8AcP0FVP+8
PQm/g/aXf2QQkQT+omdHVEJrAjEAy0pL0EBH6EVS98evDCBtQw22OZT52qXlAwZ2
gyTriKFVoqjeEjt3SZKKqXHSApP/AjBLpF99zcJJZRq2abgYlf9lv1chkrWqDHUu
DZttmYJeEfiFBBavVYIF1dOlZT0G8jMCMBc7sOSZodFnAiryP+Qg9otSBjJ3bQML
pSTqy7c3a2AScC/YyOwkDaICHnnD3XyjMwIxALRzl0tQEKMXs6hH8ToUdlLROCrP
EhQ0wahUTCk1gKA4uPD6TMTChavbh4K63OvbKg==
-----END RSA PRIVATE KEY-----"""

class ClientUserAuth(userauth.SSHUserAuthClient):

    def getPassword(self, prompt = None):
        return
        # this says we won't do password authentication

    def getPublicKey(self):
        return keys.getPublicKeyString(data = publicKey)

    def getPrivateKey(self):
        return defer.succeed(keys.getPrivateKeyObject(data = privateKey))
```

Again, fairly simple. The `SSHUserAuthClient` takes care of most of the work, but the actual authentication data needs to be supplied. `getPassword()` asks for a password, `getPublicKey()` and `getPrivateKey()` get public and private keys, respectively. `getPassword()` returns a `Deferred` that is called back with the password to use. `getPublicKey()` returns the SSH key data for the public key to use. `keys.getPublicKeyString()` will take keys in OpenSSH and LSH format, and convert them to the required format. `getPrivateKey()` returns a `Deferred` which is called back with the key object (as used in PyCrypto) for the private key. `getPassword()` and `getPrivateKey()` return `Deferreds` because they may need to ask the user for input.

Once the authentication is complete, `SSHUserAuthClient` takes care of starting the code `SSHConnection` object given to it. Next, we'll look at how to use the `SSHConnection`

9.4.5 The Connection

```
from twisted.conch.ssh import connection

class ClientConnection(connection.SSHConnection):

    def serviceStarted(self):
        self.openChannel(CatChannel(conn = self))
```

`SSHConnection` is the easiest, as it's only responsible for starting the channels. It has other methods, those will be examined when we look at `SSHChannel`.

9.4.6 The Channel

```
from twisted.conch.ssh import channel, common

class CatChannel(channel.SSHChannel):

    name = 'session'

    def channelOpen(self, data):
        d = self.conn.sendRequest(self, 'exec', common.NS('cat'),
                                  wantReply = 1)
        d.addCallback(self._cbSendRequest)
        self.catData = ''

    def _cbSendRequest(self, ignored):
        self.write('This data will be echoed back to us by "cat."\r\n')
        self.conn.sendEOF(self)
        self.loseConnection()

    def dataReceived(self, data):
        self.catData += data

    def closed(self):
        print 'We got this from "cat":', self.catData
```

Now that we've spent all this time getting the server and client connected, here is where that work pays off. `SSHChannel` is the interface between you and the other side. This particular channel opens a session and plays with the 'cat' program, but your channel can implement anything, so long as the server supports it.

The `channelOpen()` method is where everything gets started. It gets passed a chunk of data; however, this chunk is usually nothing and can be ignored. Our `channelOpen()` initializes our channel, and sends a request to the other side, using the `sendRequest()` method of the `SSHConnection` object. Requests are used to send events to the other side. We pass the method self so that it knows to send the request for this channel. The 2nd argument of 'exec' tells the server that we want to execute a command. The third argument is the data that accompanies the request. `common.NS` encodes the data as a length-prefixed string, which is how the server expects the data. We also say that we want a reply saying that the process has a been started. `sendRequest()` then returns a `Deferred` which we add a callback for.

Once the callback fires, we send the data. `SSHChannel` supports the `twisted.internet.interface.Transport` interface, so it can be given to Protocols to run them over the secure connection. In our case, we just write the data directly. `sendEOF()` does not follow the interface, but Conch uses it to tell the other side that we will write no more data. `loseConnection()` shuts down our side of the connection, but we will still receive data through `dataReceived()`. The `closed()` method is called when both sides of the connection are closed, and we use it to display the data we received (which should be the same as the data we sent.)

Finally, let's actually invoke the code we've set up.

9.4.7 The main() function

```
from twisted.internet import protocol, reactor

def main():
    factory = protocol.ClientFactory()
    factory.protocol = ClientTransport
    reactor.connectTCP('localhost', 22, factory)
    reactor.run()

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

We call `connectTCP()` to connect to localhost, port 22 (the standard port for ssh), and pass it an instance of `twisted.internet.protocol.ClientFactory`. This instance has the attribute `protocol` set to our earlier `ClientTransport` class. Note that the protocol is set to the class `ClientFactory`, not an instance of `ClientFactory`! When the `connectTCP` call completes, the protocol will be called to create a `ClientTransport()` object - this then invokes all our previous work.

It's worth noting that in the example `main()` routine, the `reactor.run()` call never returns. If you want to make the program exit, call `reactor.stop()` in the earlier `closed()` method.

If you wish to observe the interactions in more detail, adding a call to `log.startLogging(sys.stdout, setStdout=0)` before the `reactor.run()` call will send all logging to stdout.

9.5 Overview of Twisted IM

> **Note:**Twisted IM and Twisted Words are both known to be in a state of flux at the moment. Several of the APIs discussed here have fallen short of their original goals and *will* be changing within the next few releases of Twisted. The good news is that newer versions will be based on our experiences with the current ones and will provide much more access to features beyond plain-text chat message relaying in different protocols.

Twisted IM (Instance Messenger) is a multi-protocol chat framework, based on the Twisted framework we've all come to know and love. It's fairly simple and extensible in two directions - it's pretty easy to add new protocols, and it's also quite easy to add new front-ends.

9.5.1 Code flow

Twisted IM is usually started from the file `twisted/scripts/im.py` (maybe with a shell-script wrapper or similar). Twisted currently comes with two interfaces for Twisted IM - one written in GTK for Python under Linux, and one written in Swing for Jython. `im.py` picks an implementation and starts it - if you want to write your own interface, you can modify `im.py` to start it under appropriate conditions.

Once started, both interfaces behave in a very similar fashion, so I won't be getting into differences here.

AccountManager

Control flow starts at the relevant subclass of `baseaccount.AccountManager`. The AccountManager is responsible for, well, managing accounts - remembering what accounts are available, their settings, adding and removal of accounts, and making accounts log on at startup.

This would be a good place to start your interface, load a list of accounts from disk and tell them to login. Most of the method names in `AccountManager` are pretty self-explanatory, and your subclass can override whatever it wants, but you *need* to override `__init__`. Something like this:

```
def __init__(self):
self.chatui = ... # Your subclass of basechat.ChatUI
self.accounts = ... # Load account list
for a in self.accounts:
a.logOn(self.chatui)
```

ChatUI

Account objects talk to the user via a subclass of `basechat.ChatUI`. This class keeps track of all the various conversations that are currently active, so that when an account receives and incoming message, it can put that message in its correct context.

How much of this class you need to override depends on what you need to do. You will need to override `getConversation` (a one-on-one conversation, like an IRC DCC chat) and `getGroupConversation` (a multiple user conversation, like an IRC channel). You might want to override `getGroup` and `getPerson`.

The main problem with the default versions of the above routines is that they take a parameter, `Class`, which defaults to an abstract implementation of that class - for example, `getConversation` has a `Class` parameter that defaults to `basechat.Conversation` which raises a lot of `NotImplementedErrors`. In your subclass, override the method with a new method whose Class parameter defaults to your own implementation of `Conversation`, that simply calls the parent class' implementation.

Conversation and GroupConversation

These classes are where your interface meets the chat protocol. Chat protocols get a message, find the appropriate `Conversation` or `GroupConversation` object, and call its methods when various interesting things happen.

Override whatever methods you want to get the information you want to display. You must override the `hide` and `show` methods, however - they are called frequently and the default implementation raises `NotImplementedError`.

Accounts

An account is an instance of a subclass of `basesupport.AbstractAccount`. For more details and sample code, see the various `*support` files in `twisted.im`.

Chapter 10

Working on the Twisted Code Base

10.1 Twisted Coding Standard

10.1.1 Naming

Try to choose names which are both easy to remember and meaningful. Some silliness is OK at the module naming level (see `twisted.spread`...) but when choosing class names, be as precise as possible. Write code with a dictionary and thesaurus open on the table next to you.

Try to avoid overloaded terms. This rule is often broken, since it is incredibly difficult, as most normal words have already been taken by some other software. More importantly, try to avoid meaningless words. In particular, words like "handler", "processor", "engine", "manager" and "component" don't really indicate what something does, only that it does *something*.

Use American spelling in both names and docstrings. For compound technical terms such as 'filesystem', use a non-hyphenated spelling in both docstrings and code in order to avoid unnecessary capitalization.

10.1.2 Testing

Unit tests are written using the `twisted.trial` framework. Many examples are in the `twisted.test` package. Test modules should start with 'test_' in their name. Source files should have *`test-case-name`* (page 344) tags that point to their related tests.

Acceptance tests are all automated by the admin/accepttests script currently. (TODO: real acceptance tests strategy!)

Run the unit tests tests before you check anything in.

Let me repeat that, for emphasis: *run the unit tests before you check* anything *in*. Code which breaks functionality is unfortunate and unavoidable. The acceptance tests are highly nonportable and sometimes a pain to run, so this is pardonable. Code which breaks the unit tests in a way that you could have prevented by running them yourself, however, may be grounds for anything from merciless taunting through revertion of the breakage to revocation of cvs commit privileges.

It is strongly suggested that developers learn to use Emacs, and use the `twisted-dev.el` file included in the TwistedEmacs package to bind the F9 key to "run unit tests" and bang on it frequently. Support for other editors is unavailable at this time but we would love to provide it.

If you modify, or write a new, HOWTO, please read the *Lore documentation* (page 318) to learn the format the docs.

10.1.3 Whitespace

Indentation is 4 spaces per indent. Tabs are not allowed. It is preferred that every block appear on a new line, so that control structure indentation is always visible.

10.1.4 Modules

Modules must be named in all lower-case, preferably short, single words. If a module name contains multiple words, they may be separated by underscores or not separated at all.

In most cases, modules should contain more than one class, function, or method; if a module contains only one object, consider refactoring to include more related functionality in that module.

Depending on the situation, it is acceptable to have imports that look like this:

```
from twisted.internet.defer import Deferred
```

or like this:

```
from twisted.internet import defer
```

That is, modules should import *modules* or *classes and functions*, but not *packages*.

10.1.5 Packages

Package names should follow the same conventions as module names. All modules must be encapsulated in some package. Nested packages may be used to further organize related modules.

`__init__.py` must never contain anything other than a docstring and (optionally) an `__all__` attribute. Packages are not modules and should be treated differently. This rule may be broken to preserve backwards compatibility if a module is made into a nested package as part of a refactoring.

If you wish to promote code from a module to a package, for example, to break a large module out into several smaller files, the accepted way to do this is to promote from within the module. For example,

```
# parent/
# --- __init__.py ---
import child

# --- child.py ---
import parent
class Foo:
    pass
parent.Foo = Foo
```

Every package should be added to the list in `setup.py`.

Packages must not depend circularly upon each other. To simplify maintaining this state, packages must also not import each other circularly. While this applies to all packages within Twisted, one `twisted.python` deserves particular attention, as it may not depend on any other Twisted package.

10.1.6 Docstrings

Wherever possible, docstrings should be used to describe the purpose of methods, functions, classes, and modules. In cases where it's desirable to avoid documenting thoroughly – for example, and evolving interface – insert a placeholder docstring (`"UNDOCUMENTED"` is preferred), so that the auto-generated API documentation will not pick up an extraneous comment as the documentation for that module/class/function.

Docstrings are *never* to be used to provide semantic information about an object; this rule may be violated if the code in question is to be used in a system where this is a requirement (such as Zope).

Docstrings should be indented to the level of the code they are documenting.

Docstrings should be triple-quoted.

Docstrings should be written in epytext format; more documentation is available in the Epytext Markup Language documentation[1].

Additionally, to accommodate emacs users:

- Single quotes of the type of the docstring's triple-quote should be escaped. This will prevent font-lock from accidentally fontifying large portions of the file as a string.
- Code examples in docstrings should be prefixed by the — character. This will prevent IM-Python from regarding sample code as real functions, methods, and classes.

For example,

```
def foo2bar(f):
    """I am a function to convert foos to bars.

    I should be used when you have a foo but you want a bar; note that this is
    a non-destructive operation.  If I can\'t convert the foo to a bar I will
    raise a FooException().

    For example::

      |   import wombat
      |   def sample(something):
      |       f = something.getFoo()
      |       f.doFooThing()
      |       b = wombat.foo2bar(f)
      |       b.doBarThing()
      |       return b

    """
    # Optionally, actual code can go here.
```

10.1.7 Scripts

For each "script", that is, a program you expect a Twisted user to run from the command-line, the following things must be done:

[1] http://epydoc.sourceforge.net/epytext.html

1. Write a module in `twisted.scripts` which contains a callable global named `run`. This will be called by the command line part with no arguments (it will usually read `sys.argv`). Feel free to write more functions or classes in this module, if you feel they are useful to others.
2. Write a file in `bin/` which contains the Twisted running-from-CVS header, and ending with

```
from twisted.scripts.yourmodule import run
run()
```

3. Write a manpage in `doc/man`. On debian systems you can find a skeleton example of a manpage in `/usr/share/doc/man-db/examples/manpage.example`.
4. Add your script to the script list in `setup.py`.

This will insure your program will work correctly for users of CVS, Windows releases and Debian packages.

10.1.8 Standard Library Extension Modules

When using the extension version of a module for which there is also a Python version, place the import statement inside a try/except block, and import the Python version if the import fails. This allows code to work on platforms where the extension version is not available. For example:

```
try:
    import cPickle as pickle
except ImportError:
    import pickle
```

Use the "as" syntax of the import statement as well, to set the name of the extension module to the name of the Python module.

10.1.9 ChangeLog

All changes that will affect the way end-users see Twisted should come with an appropriate entry in the ChangeLog that summarizes that impact.

The correct format for the ChangeLog is GNU changelog format. There is an emacs mode for editing this, use `M-x add-change-log-entry`. If you are, for whatever absurd reason, using an editor other than emacs to edit Twisted, you can use Moshe Zadka's helpfully provided `admin/change` script to add a properly-formatted entry.

10.1.10 Classes

Classes are to be named in mixed case, with the first letter capitalized; each word separated by having its first letter capitalized. Acronyms should be capitalized in their entirety. Class names should not be prefixed with the name of the module they are in. Examples of classes meeting this criteria:

- twisted.spread.pb.ViewPoint
- twisted.parser.patterns.Pattern

Examples of classes *not* meeting this criteria:

- event.EventHandler
- main.MainGadget

An effort should be made to prevent class names from clashing with each other between modules, to reduce the need for qualification when importing. For example, a Service subclass for Forums might be named twisted.forum.service.ForumService, and a Service subclass for Words might be twisted.words.service.WordsService. Since neither of these modules are volatile *(see above)* the classes may be imported directly into the user's namespace and not cause confusion.

10.1.11 Methods

Methods should be in mixed case, with the first letter lower case, each word separated by having its first letter capitalized. For example, `someMethodName`, `method`.

Sometimes, a class will dispatch to a specialized sort of method using its name; for example, twisted.reflect.Accessor. In those cases, the type of method should be a prefix in all lower-case with a trailing underscore, so method names will have an underscore in them. For example, `get_someAttribute`. Underscores in method names in twisted code are therefore expected to have some semantic associated with them.

Some methods, in particular `addCallback` and its cousins return self to allow for chaining calls. In this case, wrap the chain in parenthesis, and start each chained call on a separate line, for example:

```
return (foo()
        .addCallback(bar)
        .addCallback(thud)
        .addCallback(wozers))
```

10.1.12 Callback Arguments

There are several methods whose purpose is to help the user set up callback functions, for example `Deferred.addCallback` or the reactor's `callLater` method. To make access to the callback as transparent as possible, most of these methods use `**kwargs` to capture arbitrary arguments that are destined for the user's callback. This allows the call to the setup function to look very much like the eventual call to the target callback function.

In these methods, take care to not have other argument names that will "steal" the user's callback's arguments. When sensible, prefix these "internal" argument names with an underscore. For example, `RemoteReference.callRemote` is meant to be called like this:

```
myref.callRemote("addUser", "bob", "555-1212")

# on the remote end, the following method is invoked:
def addUser(name, phone):
    ...
```

where "addUser" is the remote method name. The user might also choose to call it with named parameters like this:

```
myref.callRemote("addUser", name="bob", phone="555-1212")
```

In this case, `callRemote` (and any code that uses the **kwargs syntax) must be careful to not use "name", "phone", or any other name that might overlap with a user-provided named parameter. Therefore, `callRemote` is implemented with the following signature:

```
def callRemote(self, _name, *args, **kw):
    ...
```

Do whatever you can to reduce user confusion. It may also be appropriate to `assert` that the kwargs dictionary does not contain parameters with names that will eventually cause problems.

10.1.13 Special Methods

The augmented assignment protocol, defined by __iadd__ and other similarly named methods, can be used to allow objects to be modified in place or to rebind names if an object is immutable – both through use of the same operator. This can lead to confusing code, which in turn leads to buggy code. For this reason, methods of the augmented assignment protocol should not be used in Twisted.

10.1.14 Functions

Functions should be named similiarly to methods.

Functions or methods which are responding to events to complete a callback or errback should be named `_cbMethodName` or `_ebMethodName`, in order to distinguish them from normal methods.

10.1.15 Attributes

Attributes should be named similarly to functions and methods. Attributes should be named descriptively; attribute names like `mode`, `type`, and `buf` are generally discouraged. Instead, use `displayMode`, `playerType`, or `inputBuffer`.

Do not use Python's "private" attribute syntax; prefix non-public attributes with a single leading underscore. Since several classes have the same name in Twisted, and they are distinguished by which package they come from, Python's double-underscore name mangling will not work reliably in some cases. Also, name-mangled private variables are more difficult to address when unit testing or persisting a class.

An attribute (or function, method or class) should be considered private when one or more of the following conditions are true:

- The attribute represents intermediate state which is not always kept up-to-date.
- Referring to the contents of the attribute or otherwise maintaining a reference to it may cause resources to leak.
- Assigning to the attribute will break internal assumptions.
- The attribute is part of a known-to-be-sub-optimal interface and will certainly be removed in a future release.

10.1.16 Database

Database tables will be named with plural nouns.

Database columns will be named with underscores between words, all lower case, since most databases do not distinguish between case.

Any attribute, method argument, or method name that corresponds *directly* to a column in the database will be named exactly the same as that column, regardless of other coding conventions surrounding that circumstance.

All SQL keywords should be in upper case.

10.1.17 C Code

Wherever possible, C code should be optional, and the default python implementation should be maintained in tandem with it. C code should be strict ANSI C, and *must* build using GCC as well as Visual Studio for Windows, and really shouldn't have any problems with other compilers either. Don't do anything tricky.

C code should only be used for efficiency, not for binding to external libraries. If your particular code is not frequently run, write it in Python. If you require the use of an external library, develop a separate, external bindings package and make your twisted code depend on it.

10.1.18 Checkin Messages

Thanks to CVSToys, the checkin messages are being used in a myriad of ways. Because of that, you need to observe a few simple rules when writing a checkin message.

The first line of the message is being used as both the subject of the commit e-mail and the announcement on #twisted. Therefore, it should be short (aim for < 80 characters) and descriptive – and must be able to stand alone (it is best if it is a complete sentence). The rest of the e-mail should be separated with *hard line breaks* into short lines (< 70 characters). This is free-format, so you can do whatever you like here.

Checkin messages should be about *what*, not *how*: we can get how from CVS diff. Explain reasons for checkins, and what they affect.

Each commit should be a single logical change, which is internally consistent. If you can't summarize your changes in one short line, this is probably a sign that they should be broken into multiple checkins.

10.1.19 Recommendations

These things aren't necessarily standardizeable (in that code can't be easily checked for compliance) but are a good idea to keep in mind while working on Twisted.

If you're going to work on a fragment of the Twisted codebase, please consider finding a way that you would *use* such a fragment in daily life. I use the Twisted Web server on the main TML website, and aside from being good PR, this encourages you to actively maintain and improve your code, as the little everyday issues with using it become apparent.

Twisted is a *big* codebase! If you're refactoring something, please make sure to recursively grep for the names of functions you're changing. You may be surprised to learn where something is called. Especially if you are moving or renaming a function, class, method, or module, make sure that it won't instantly break other code.

10.2 HTML Documentation Standard for Twisted

10.2.1 Allowable Tags

Please try to restrict your HTML usage to the following tags (all only for the original logical purpose, and not whatever visual effect you see): `<html>`, `<title>`, `<head>`, `<body>`, `<h1>`, `<h2`, `<h3>`, `<ol>`, `<ul>`, `<dl>`, `<li>`,

`<dt>`, `<dd>`, `<p>`, `<code>`, `<img>`, `<blockquote>`, `<a>`, `<cite>`, `<div>`, `<span>`, `<strong>`, `<em>`, `<pre>`, `<q>`, `<table>`,`<tr>`, `<td>` and `<th>`.

Please avoid using the quote sign (`"`) for quoting, and use the relevant html tags (`<q></q>`) – it is impossible to distinguish right and left quotes with the quote sign, and some more sophisticated output methods work better with that distinction.

10.2.2 Multi-line Code Snippets

Multi-line code snippets should be delimited with a <pre> tag, with a mandatory “class” attribute. The conventionalized classes are “python”, “python-interpreter”, and “shell”. For example:

“python”

```
    <p>
    For example, this is how one defines a Resource:
    </p>

    <pre class="python">
from twisted.web import resource

class MyResource(resource.Resource):
    def render_GET(self, request):
        return "Hello, world!"
    </pre>
```

For example, this is how one defines a Resource:

```
from twisted.web import resource

class MyResource(resource.Resource):
    def render_GET(self, request):
        return "Hello, world!"
```

Note that you should never have leading indentation inside a <pre> block – this makes it hard for readers to copy/paste the code.

“python-interpreter”

```
    <pre class="python-interpreter">
    \&gt;\&gt;\&gt; from twisted.web import resource
    \&gt;\&gt;\&gt; class MyResource(resource.Resource):
    ...     def render_GET(self, request):
    ...         return "Hello, world!"
    ...
    \&gt;\&gt;\&gt; MyResource().render_GET(None)
    "Hello, world!"
    </pre>
```

```
>>> from twisted.web import resource
>>> class MyResource(resource.Resource):
...     def render_GET(self, request):
...         return "Hello, world!"
...
>>> MyResource().render_GET(None)
"Hello, world!"
```

“shell”

```
<pre class="shell">
$ mktap web --path /var/www
</pre>
```

```
$ mktap web --path /var/www
```

10.2.3 Code inside paragraph text

For single-line code-snippets and attribute, method, class, and module names, use the <code> tag, with a class of “API” or “python”. During processing, module or class-names with class “API” will automatically be looked up in the API reference and have a link placed around it referencing the actual API documents for that module/classname. If you wish to reference an API document, then make sure you at least have a single module-name so that the processing code will be able to figure out which module or class you’re referring to.

You may also use the `base` attribute in conjuction with a class of “API” to indicate the module that should be prepended to the module or classname. This is to help keep the documentation clearer and less cluttered by allowing links to API docs that don’t need the module name.

```
    <p>
To add a <code class="API">twisted.web.widgets.Widget</code>
instance to a <code class="API"
base="twisted.web.widgets">Gadget</code> instance, do
<code class="python">myGadget.putWidget("widgetPath",
MyWidget())</code>.
    </p>

    <p>
 (implementation note: the widgets are stored in the <code
class="python">gadgetInstance.widgets</code> attribute,
which is a
list.)
    </p>
```

To add a `twisted.web.widgets.Widget` instance to a `Gadget` instance, do `myGadget.putWidget("widgetPath", MyWidget())`.

(implementation note: the widgets are stored in the `gadgetInstance.widgets` attribute, which is a list.)

10.2.4 Headers

It goes without mentioning that you should use <hN> in a sane way – <h1> should only appear once in the document, to specify the title. Sections of the document should use <h2>, sub-headers <h3>, and so on.

10.2.5 XHTML

XHTML is mandatory. That means tags that don't have a closing tag need a "/"; for example, `<hr />` . Also, tags which have "optional" closing tags in HTML *need* to be closed in XHTML; for example, `<li>foo</li>`

10.2.6 Tag Case

All tags will be done in lower-case. XHTML demands this, and so do I. :-)

10.2.7 Footnotes

Footnotes are enclosed inside `<span class="footnote"></span>`. They must not contain any markup.

10.2.8 Suggestions

Use `lore -o lint` to check your documentation is not broken. `lore -o lint` will never change your HTML, but it will complain if it doesn't like it.

Don't use tables for formatting. 'nuff said.

10.2.9 __all__

`__all__` is a module level list of strings, naming objects in the module that are public. Make sure publically exported classes, functions and constants are listed here.

10.3 Unit Tests in Twisted

Each *unit test* tests one bit of functionality in the software. Unit tests are entirely automated and complete quickly. Unit tests for the entire system are gathered into one test suite, and may all be run in a single batch. The result of a unit test is simple: either it passes, or it doesn't. All this means you can test the entire system at any time without inconvenience, and quickly see what passes and what fails.

10.3.1 Unit Tests in the Twisted Philosophy

The Twisted development team adheres to the practice of Extreme Programming[2] (XP), and the usage of unit tests is a cornerstone XP practice. Unit tests are a tool to give you increased confidence. You changed an algorithm – did you break something? Run the unit tests. If a test fails, you know where to look, because each test covers only a small amount of code, and you know it has something to do with the changes you just made. If all the tests pass, you're good to go, and you don't need to second-guess yourself or worry that you just accidently broke someone else's program.

[2] http://c2.com/cgi/wiki?ExtremeProgramming

10.3.2 What to Test, What Not to Test

You don't have to write a test for every single method you write, only production methods that could possibly break.
– Kent Beck, Extreme Programming Explained, p. 58.

10.3.3 Running the Tests

How

```
$ Twisted/admin/runtests
```

You'll find that having something like this in your emacs init files is quite handy:

```
(defun runtests () (interactive)
  (compile "python /somepath/Twisted/admin/runtests"))

(global-set-key [(alt t)] 'runtests)
```

When

Always always *always* be sure all the tests pass[3] before committing any code. If someone else checks out code at the start of a development session and finds failing tests, they will not be happy and may decide to *hunt you down*.

Since this is a geographically dispersed team, the person who can help you get your code working probably isn't in the room with you. You may want to share your work in progress over the network, but you want to leave the main CVS tree in good working order. So use a branch[4], and merge your changes back in only after your problem is solved and all the unit tests pass again.

10.3.4 Adding a Test

Please don't add new modules to Twisted without adding tests for them too. Otherwise we could change something which breaks your module and not find out until later, making it hard to know exactly what the change that broke it was, or until after a release, and nobody wants broken code in a release.

Tests go in Twisted/twisted/test/, and are named `test_foo.py`, where `foo` is the name of the module or package being tested. Extensive documentation on using the PyUnit framework for writing unit tests can be found in the *links section* (page 350) below.

One deviation from the standard PyUnit documentation: To ensure that any variations in test results are due to variations in the code or environment and not the test process itself, Twisted ships with its own, compatible, testing framework. That just means that when you import the unittest module, you will `from twisted.trial import unittest` instead of the standard `import unittest`.

As long as you have followed the module naming and placement conventions, `runtests` will be smart enough to pick up any new tests you write.

[3] http://www.xprogramming.com/xpmag/expUnitTestsAt100.htm
[4] http://www.cvshome.org/docs/manual/cvs_5.html

10.3.5 Skipping tests, TODO items

Trial, the Twisted unit test framework, has some extensions which are designed to encourage developers to add new tests. One common situation is that a test exercises some optional functionality: maybe it depends upon certain external libraries being available, maybe it only works on certain operating systems. The important common factor is that nobody considers these limitations to be a bug.

To make it easy to test as much as possible, some tests may be skipped in certain situations. Individual test cases can raise the `SkipTest` exception to indicate that they should be skipped, and the remainder of the test is not run. In the summary (the very last thing printed, at the bottom of the test output) the test is counted as a "skip" instead of a "success" or "fail". This should be used inside a conditional which looks for the necessary prerequisites:

```
def testSSHClient(self):
    if not ssh_path:
        raise unittest.SkipTest, "cannot find ssh, nothing to test"
    foo() # do actual test after the SkipTest
```

You can also set the .skip attribute on the method, with a string to indicate why the test is being skipped. This is convenient for temporarily turning off a test case, but it can also be set conditionally (by manipulating the class attributes after they've been defined):

```
def testThing(self):
    dotest()
testThing.skip = "disabled locally"

class MyTestCase(unittest.TestCase):
    def testOne(self):
        ...
    def testThing(self):
        dotest()

if not haveThing:
    MyTestCase.testThing.im_func.skip = "cannot test without Thing"
    # but testOne() will still run
```

Finally, you can turn off an entire TestCase at once by setting the .skip attribute on the class. If you organize your tests by the functionality they depend upon, this is a convenient way to disable just the tests which cannot be run.

```
class SSLTestCase(unittest.TestCase):
   ...
class TCPTestCase(unittest.TestCase):
   ...

if not haveSSL:
    SSLTestCase.skip = "cannot test without SSL support"
    # but TCPTestCase will still run
```

.todo and Testing New Functionality

Two good practices which arise from the "XP" development process are sometimes at odds with each other:

- Unit tests are a good thing. Good developers recoil in horror when they see a failing unit test. They should drop everything until the test has been fixed.
- Good developers write the unit tests first. Once tests are done, they write implementation code until the unit tests pass. Then they stop.

These two goals will sometimes conflict. The unit tests that are written first, before any implementation has been done, are certain to fail. We want developers to commit their code frequently, for reliability and to improve coordination between multiple people working on the same problem together. While the code is being written, other developers (those not involved in the new feature) should not have to pay attention to failures in the new code. We should not dilute our well-indoctrinated Failing Test Horror Syndrome by crying wolf when an incomplete module has not yet started passing its unit tests. To do so would either teach the module author to put off writing or committing their unit tests until *after* all the functionality is working, or it would teach the other developers to ignore failing test cases. Both are bad things.

".todo" is intended to solve this problem. When a developer first starts writing the unit tests for functionality that has not yet been implemented, they can set the `.todo` attribute on the test methods that are expected to fail. These methods will still be run, but their failure will not be counted the same as normal failures: they will go into an "expected failures" category. Developers should learn to treat this category as a second-priority queue, behind actual test failures.

As the developer implements the feature, the tests will eventually start passing. This is surprising: after all those tests are marked as being expected to fail. The .todo tests which nevertheless pass are put into a "unexpected success" category. The developer should remove the .todo tag from these tests. At that point, they become normal tests, and their failure is once again cause for immediate action by the entire development team.

The life cycle of a test is thus:

1. Test is created, marked `.todo`. Test fails: "expected failure".
2. Code is written, test starts to pass. "unexpected success".
3. `.todo` tag is removed. Test passes. "success".
4. Code is broken, test starts to fail. "failure". Developers spring into action.
5. Code is fixed, test passes once more. "success".

Any test which remains marked with `.todo` for too long should be examined. Either it represents functionality which nobody is working on, or the test is broken in some fashion and needs to be fixed.

10.3.6 Associating Test Cases With Source Files

Please add a `test-case-name` tag to the source file that is covered by your new test. This is a comment at the beginning of the file which looks like one of the following:

```
# -*- test-case-name: twisted.test.test_defer -*-
```

or

```
#!/usr/bin/python
# -*- test-case-name: twisted.test.test_defer -*-
```

This format is understood by emacs to mark "File Variables". The intention is to accept `test-case-name` anywhere emacs would on the first or second line of the file (but not in the `File Variables:` block that emacs accepts at the end of the file). If you need to define other emacs file variables, you can either put them in the `File Variables:` block or use a semicolon-separated list of variable definitions:

```
# -*- test-case-name: twisted.test.test_defer; fill-column: 75; -*-
```

If the code is exercised by multiple test cases, those may be marked by using a comma-separated list of tests, as follows: (NOTE: not all tools can handle this yet.. trial –testmodule does, though)

```
# -*- test-case-name: twisted.test.test_defer,twisted.test.test_tcp -*-
```

The `test-case-name` tag will allow `trial --testmodule twisted/dir/myfile.py` to determine which test cases need to be run to exercise the code in `myfile.py`. Several tools (as well as `twisted-dev.el`'s F9 command) use this to automatically run the right tests.

10.3.7 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 one 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.

Your test is responsible for cleaning up after itself. The `tearDown` method is an ideal place for this cleanup code: it is always run regardless of whether your test passes or fails (like a bare `except` clause in a try-except construct). Exceptions in tearDown are flagged as errors and flunk the test.

TODO: helper functions: TestCase.addPort, TestCase.addTimer

reactor.stop is considered very harmful, and should only be used by reactor-specific test cases which know how to restore the state that it kills. If you must use reactor.run, use reactor.crash to stop it instead of reactor.stop.

Trial tries to help insure that the reactor is clean after each test, but the reactor does not yet support an interface that would make this work properly. It can catch leftover timers, but not lingering sockets.

deferredResult

If your test creates a `Deferred` and simply wants to verify something about its result, use `deferredResult`. It will wait for the Deferred to fire and give you the result. If the Deferred runs the errback handler instead, it will raise an exception so your test can fail. Note that the *only* thing that will terminate a `deferredResult` call is if the Deferred fires; in particular, timers which raise exceptions will not cause it to return.

Waiting for Things

The preferred way to run a test that waits for something to happen (always triggered by other things that you have done) is to use a `while not self.done` loop that does `reactor.iterate(0.1)` at the beginning of each pass. The "0.1" argument sets a limit on how long the reactor will wait to return if there is nothing to do. 100 milliseconds is long enough to avoid spamming the CPU while your timers wait to expire.

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 CVS repository.

Trial tests indicate they have failed by raising a FailTest exception (self.fail and friends are just wrappers around this `raise` statement). Exceptions that are raised inside a callRemote timer are caught and logged but otherwise ignored. Trial uses a logging hook to notice when errors have been logged by the test that just completed (so such errors will flunk the test), but this happens after the fact: they will not be noticed by the main body of your test code. Therefore callRemote timers can not be used directly to establish timeouts which terminate and flunk the test.

The right way to implement this sort of timeout is to have a `self.done` flag, and a while loop which iterates the reactor until it becomes true. Anything that causes the test to be finished (success *or* failure) can set self.done to cause the loop to exit.

Most of the code in Twisted is run by the reactor as a result of socket activity. This is almost always started by Protocol.connectionMade or Protocol.dataReceived (because the output side goes through a buffer which queues data for transmission). Exceptions that are raised by code called in this way (by the reactor, through doRead or doWrite) are caught, logged, handed to connectionLost, and otherwise ignored.

This means that your Protocol's connectionLost method, if invoked because of an exception, must also set this self.done flag. Otherwise the test will not terminate.

Exceptions that are raised in a Deferred callback are turned into a Failure and stashed inside the Deferred. When an errback handler is attached, the Failure is given to it. If the Deferred goes out of scope while an error is still pending, the error is logged just like exceptions that happen in timers or protocol handlers. This will cause the current test to flunk (eventually), but it is not checked until after the test fails. So again, it is a good idea to add errbacks to your Deferreds that will terminate your test's main loop.

Here is a brief example that demonstrates a few of these techniques.

```
class MyTest(unittest.TestCase):
    def setUp(self):
        self.done = False
        self.failure = None

    def tearDown(self):
        self.server.stopListening()
        # TODO: also shut down client
        try:
            self.timeout.cancel()
        except (error.AlreadyCancelled, error.AlreadyCalled):
            pass
```

```
    def succeeded(self):
        self.done = True

    def failed(self, why):
        self.done = True
        self.failure = why

    def testServer(self):
        self.server = reactor.listenTCP(port, factory)
        self.client = reactor.connectTCP(port, factory)
        # you should give the factories a way to call our 'succeeded' or
        # 'failed' methods
        self.timeout = reactor.callLater(5, self.failed, "timeout")
        while not self.done:
            reactor.iterate(0.1)

        # we get here if the test is finished, for good or for bad
        if self.failure:
            self.fail(self.failure)
        # otherwise it probably passed. Cleanup will be done in tearDown()
```

10.3.8 Links

- A chapter on Unit Testing[5] in Mark Pilgrim's Dive Into Python[6].
- `unittest`[7] module documentation, in the Python Library Reference[8].
- UnitTests[9] on the PortlandPatternRepository Wiki[10], where all the cool ExtremeProgramming[11] kids hang out.
- Unit Tests[12] in Extreme Programming: A Gentle Introduction[13].
- Ron Jeffries espouses on the importance of Unit Tests at 100%[14].
- Ron Jeffries writes about the Unit Test[15] in the Extreme Programming practices of C3[16].
- PyUnit's homepage[17].

[5] http://diveintopython.org/roman_divein.html
[6] http://diveintopython.org
[7] http://www.python.org/doc/current/lib/module-unittest.html
[8] http://www.python.org/doc/current/lib/
[9] http://c2.com/cgi/wiki?UnitTests
[10] http://c2.com/cgi/wiki
[11] http://c2.com/cgi/wiki?ExtremeProgramming
[12] http://www.extremeprogramming.org/rules/unittests.html
[13] http://www.extremeprogramming.org
[14] http://www.xprogramming.com/xpmag/expUnitTestsAt100.htm
[15] http://www.xprogramming.com/Practices/PracUnitTest.html
[16] http://www.xprogramming.com/Practices/xpractices.htm
[17] http://pyunit.sourceforge.net

- twisted.test[18]'s inline documentation.
- The twisted/test directory[19] in CVS.

10.4 Working from Twisted's Subversion repository

If you're going to be doing development on Twisted itself, or if you want to take advantage of bleeding-edge features (or bug fixes) that are not yet available in a numbered release, you'll probably want to check out a tree from the Twisted Subversion repository. The Trunk is where all current development takes place.

This document lists some useful tips for working on this cutting edge.

10.4.1 Checkout

Subversion tutorials can be found elsewhere, see in particular the Subversion homepage[20]. The relevant data you need to check out a copy of the Twisted tree is available on the pserver page[21], and is as follows:

```
$ svn co svn://svn.twistedmatrix.com/svn/Twisted/trunk Twisted
```

10.4.2 Alternate tree names

By using `cvs checkout -d foo Twisted`, you can put the workspace tree in a directory other than "Twisted". I do this (with a name like "Twisted-Subversion") to remind myself that this tree comes from Subversion and not from a released version (like "Twisted-1.0.5"). This practice can cause a few problems, because there are a few places in the Twisted tree that need to know where the tree starts, so they can add it to `sys.path` without requiring the user manually set their PYTHONPATH. These functions walk the current directory up to the root, looking for a directory named "Twisted" (sometimes exactly that, sometimes with a `.startswith` test). Generally these are test scripts or other administrative tools which expect to be launched from somewhere inside the tree (but not necessarily from the top).

If you rename the tree to something other than `Twisted`, these tools may wind up trying to use Twisted source files from /usr/lib/python2.2 or elsewhere on the default `sys.path`. Normally this won't matter, but it is good to be aware of the issue in case you run into problems.

`twisted/test/process_twisted.py` is one of these programs.

10.4.3 Compiling C extensions

There are currently 3 C extension modules in Twisted: twisted.internet.cReactor, twisted.runner.portmap, and twisted.spread.cBanana . These modules are optional, but you'll have to compile them if you want to experience their features, performance improvements, or bugs. There are two approaches.

The first is to do a regular distutils `./setup.py build`, which will create a directory under `build/` to hold both the generated `.so` files as well as a copy of the 600-odd `.py` files that make up Twisted. If you do this, you will need to set your PYTHONPATH to something like `MyDir/Twisted/build/lib.linux-i686-2.2` in order to run code against the Subversion twisted (as opposed to whatever's installed in `/usr/lib/python2.2` or

[18]http://twistedmatrix.com/documents/TwistedDocs/current/api/public/toc-twisted.test-module.html
[19]http://twistedmatrix.com/users/jh.twistd/viewcvs/cgi/viewcvs.cgi/twisted/test/?cvsroot=Twisted
[20]http://subversion.tigris.org/
[21]http://twistedmatrix.com/developers/cvs

wherever python usually looks). In addition, you will need to re-run the `build` command *every time* you change a `.py` file. The `build/lib.foo` directory is a copy of the main tree, and that copy is only updated when you re-run `setup.py build`. It is easy to forget this and then wonder why your code changes aren't being expressed.

The second technique is to build the C modules in place, and point your PYTHONPATH at the top of the tree, like `MyDir/Twisted`. This way you're using the .py files in place too, removing the confusion a forgotten rebuild could cause with the separate build/ directory above. To build the C modules in place, do `./setup.py build_ext -i`. You only need to re-run this command when you change the C files. Note that `setup.py` is not Make, it does not always get the dependencies right (`.h` files in particular), so if you are hacking on the cReactor you may need to manually delete the `.o` files before doing a rebuild. Also note that doing a `setup.py clean` will remove the `.o` files but not the final `.so` files, they must be deleted by hand.

10.4.4 Running tests

To run the full unit-test suite, do:

```
./bin/trial -v twisted.test
```

To run a single test file (like `twisted/test/test_defer.py`), do one of:

```
./bin/trial -v twisted.test.test_defer
```

or

```
./bin/trial -v twisted/test/test_defer.py
```

To run any tests that are related to a code file, like `twisted/protocols/imap4.py`, do:

```
./bin/trial -v --testmodule twisted/protocols/imap4.py
```

This depends upon the `.py` file having an appropriate "test-case-name" tag that indicates which test cases provide coverage. See the *Test Standards* (page 344) document for details about using "test-case-name". In this example, the `twisted.test.test_imap` test will be run.

Many tests create temporary files in /tmp or ./_trial_temp, but everything in /tmp should be deleted when the test finishes. Sometimes these cleanup calls are commented out by mistake, so if you see a stray /tmp/@12345.1 directory, it is probably from test_dirdbm or test_popsicle. Look for an `rmtree` that has been commented out and complain to the last developer who touched that file.

10.4.5 Admin scripts

The `admin/` directory holds several administrative tools, some of which are used when turning a Subversion checkout into a full numbered release.

10.4.6 Building docs

Twisted documentation (not including the automatically-generated API docs) is in *Lore Format* (page 318). These `.html` files are translated into `.xhtml` files by the "bin/lore" script, which can check the files for syntax problems (hlint), process multiple files at once, insert the files into a template before processing, and can also translate the files into LaTeX or PostScript instead.

To generate the full documentation set, run the `admin/process-docs` shell script. This will create processed HTML, man pages, and 250-page "book.pdf" file containing all the docs rolled into a single nicely-formatted volume. This script needs several helper tools to handle the images and the LaTeX conversion: debian packages "tetex-extra", "netpbm", and "gs-common" should be sufficient. The docs-build process currently takes about 3 minutes on the twistedmatrix.com build machine.

To build just the HTML form of the howto/ docs, do a subset of the work done in `admin/process-docs`, such as the following. Note that the index file will be placed in `doc/howto/index.xhtml`.

```
./bin/lore -p --config template=doc/howto/template.tpl doc/howto/*.html
```

To run hlint over a single Lore document, such as `doc/howto/cvs-dev.html`, do the following. This is useful because the HTML conversion may bail without a useful explanation if it sees mismatched tags.

```
./bin/lore -n --output lint doc/howto/cvs-dev.html
```

To convert it to HTML (including markup, interpolation of examples, footnote processing, etc), do the following. The results will be placed in `doc/howto/cvs-dev.xhtml`:

```
./bin/lore -p --config template=doc/howto/template.tpl doc/howto/cvs-dev.html
```

Note that hyperlinks to other documents may not be quite right unless you include a "-l" argument to `bin/lore`. Links in the .html file are to .html targets: when the .html is turned into .xhtml, the link targets are supposed to be turned into .xhtml also. In addition to this, Lore markup of the form <code class="API"> is supposed to turn into a link to the corresponding API reference page. These links will probably be wrong unless the correct base URL is provided to Lore.

10.4.7 Emacs

Check out the TwistedEmacs module (which lives in the same Subversion repository, just do "cvs checkout TwistedEmacs" instead of "cvs checkout Twisted"). `twisted-dev.el` has several utility functions which make it easier to grep for methods, run test cases, process Lore documents, etc.

10.4.8 Building Debian packages

Running "debuild -uc -us" from the top of the Subversion tree will (hopefully) result in a collection of .deb packages in the tree's parent directory. This requires other tools to be installed (devscripts for one), and requires that "admin/process-docs" be run first. The .debs created will have a version number based upon whatever is at the top of `debian/changelog`, which is generally only updated when an official release is made, so be careful that you don't create confusingly-numbered package files.

Chapter 11

Manual Pages

11.1 CONCH.1

11.1.1 NAME

conch - connect to SSH servers

11.1.2 SYNOPSIS

conch [-l *user*] [-i *identity* [-i *identity* ...]] [-c *cipher*] [-m *MAC*] [-p *port*] [-n] [-t] [-T] [-V] [-C] [-N] [-s] [arg [...]]
conch –help

11.1.3 DESCRIPTION

The *–help* prints out a usage message to standard output.

-t*, *–user<**user**> User name to use

-i*, *–identity<**identity**> Add an identity file.

-c*, *–cipher<**cipher**> Cipher algorithm to use.

-m*, *–macs<**mac**> Specify MAC algorithms for protocol version 2.

-p*, *–port<**port**> Port to connect to.

-n*, *–null Redirect input from /dev/null

-t*, *–tty Allocate a tty even if command is given.

-n*, *–notty Do not allocate a tty.

-V*, *–version Display version number only.

-C*, *–compress nable compression.

-N, –noshell Do not execute a shell or command.

-s, –subsystem Invoke command (mandatory) as SSH2 subsystem

–log Log to stderr

11.1.4 DESCRIPTION

Open an SSH connection to specified server, and either run the command given there or open a remote interactive shell.

11.1.5 AUTHOR

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

11.1.6 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

11.1.7 COPYRIGHT

11.1.8 SEE ALSO

ssh(1)

11.2 GENERATELORE.1

11.2.1 NAME

lore - convert documentations formats

11.2.2 SYNOPSIS

lore [-l *linkrel*] [-d *docsdir*] [-i *input*] [-o *output*] [–config attribute[=value] [...]] [-p] [file [...]]
lore –help

11.2.3 DESCRIPTION

The *–help* prints out a usage message to standard output.

-p*, *–plain Use non-flashy progress bar - one file per line.

-n*, *–null Do not report progress at all.

-l*, *–linkrel Where non-document links should be relative to.

-d*, *–docsdir Where to look for *.html* files if no files are given.

-i*, *–input Input format. New input formats can be dynamically registered. Lore itself comes with "lore" (the standard format), "mlore" (allows LaTeX equations) and "man" (man page format). If the input format is not registered as a plugin, a module of the named input will be searched. For example, *–i* twisted.lore.defaultis equivalent to using the default Lore input.

-o*, *–output Output format. Available output formats depend on the input. For the core formats, lore and mlore support html, latex and lint, while man allows lore.

–config Add input/output-specific information. HTML output allows for 'ext=<extension>', 'template=<template>' and 'baseurl=<format string for API URLs>'. LaTeX output allows for 'section' or 'chapter' in Lore, and nothing in Math-Lore. Lore output allows for 'ext=<extension>'. Lint output allows nothing. Note that disallowed *–config* options are merely ignored, and do not cause errors.

11.2.4 DESCRIPTION

If no files are given, all *.html documents in docsdir are processed.

11.2.5 SEE ALSO

doc/howto/lore.xhtml, doc/howto/doc-standard.xhtml, doc/howto/extend-lore.xhtml

11.2.6 AUTHOR

Written by Moshe Zadka

11.2.7 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

11.2.8 COPYRIGHT

Copyright ©2003 Matthew W. Lefkowitz This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

11.3 IM.1

11.3.1 NAME

im - run Instance Messenger, the Tkinter twisted.words client

11.3.2 SYNOPSIS

im

11.3.3 DESCRIPTION

Run Instance Messenger, the Tkinter twisted.words client

11.3.4 AUTHOR

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

11.3.5 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

11.3.6 COPYRIGHT

Copyright ©2000 Matthew W. Lefkowitz This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

11.4 MANHOLE.1

11.4.1 NAME

manhole - Connect to a Twisted Manhole service

11.4.2 SYNOPSIS

manhole

11.4.3 DESCRIPTION

manhole is a GTK interface to Twisted Manhole services. You can execute python code as if at an interactive Python console inside a running Twisted process with this.

11.4.4 AUTHOR

Written by Chris Armstrong, copied from Moshe Zadka's "faucet" manpage.

11.4.5 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

11.4.6 COPYRIGHT

Copyright ©2000 Matthew W. Lefkowitz This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

11.5 MKTAP.1

11.5.1 NAME

mktap - create twisted.servers

11.5.2 SYNOPSIS

mktap [*options*] *apptype* [*application_option*]...
mktapapptype –help

11.5.3 DESCRIPTION

The *–help* prints out a usage message to standard output.

***–uid*, *-u*<*uid*>** Application belongs to this uid, and should run with its permissions.

***–gid*, *-d*<*gid*>** Application belongs to this gid, and should run with its permissions.

***–append*, *-a*<*file*>** Append given servers to given file, instead of creating a new one. File should be be a tap file.

***–appname*, *-n*<*name*>** Use the specified name as the process name when the application is run with *twistd(1)*. This option also causes some initialization code to be duplicated when *twistd(1)* is run.

–xml*, *-x Output as a .tax XML file rather than a pickle.

–source*, *-s Output as a .tas (AOT Python source) file rather than a pickle.

apptype Can be 'web', 'portforward', 'toc', 'coil', 'words', 'manhole', 'im', 'news', 'socks', 'telnet', 'parent', 'sibling', 'ftp', and 'mail'. Each of those support different options.

11.5.4 portforward options

***-h*, *–host*<*host*>** Proxy connections to <*host*>

***-d*, *–dest_port*<*port*>** Proxy connections to <*port*> on remote host.

***-p*, *–port*<*port*>** Listen locally on <*port*>

11.5.5 web options

-u*, *–user Makes a server with ˜/public_html and ˜/.twistd-web-pb support for users.

–personal Instead of generating a webserver, generate a ResourcePublisher which listens on ˜/.twistd-web-pb

***–path*<*path*>** <path> is either a specific file or a directory to be set as the root of the web server. Use this if you have a directory full of HTML, cgi, php3, epy, or rpy files or any other files that you want to be served up raw.

***-p*, *–port*<*port*>** <port> is a number representing which port you want to start the server on.

-m*, *–mime_type<mimetype> <mimetype> is the default MIME type to use for files in a –path web server when none can be determined for a particular extension. The default is 'text/html'.

–allow_ignore_ext Specify whether or not a request for 'foo' should return 'foo.ext'. Default is off.

–ignore-ext.<extension> Specify that a request for 'foo' should return 'foo.*<extension>*'.

-t*, *–telnet<port> Run a telnet server on <port>, for additional configuration later.

-i*, *–index<name> Use an index name other than "index.html"

–https<port> Port to listen on for Secure HTTP.

-c*, *–certificate<filename> SSL certificate to use for HTTPS. [default: server.pem]

-k*, *–privkey<filename> SSL certificate to use for HTTPS. [default: server.pem]

–processor<ext>=<class name> Adds a processor to those file names. (Only usable if after *–path)*

–resource-script<script name> Sets the root as a resource script. This script will be re-evaluated on every request.

This creates a web.tap file that can be used by twistd. If you specify no arguments, it will be a demo webserver that has the Test class from twisted.web.test in it.

11.5.6 toc options

-p<port> <port> is a number representing which port you want to start the server on.

11.5.7 mail options

-r*, *–relay<ip>,<port>=<queue directory> Relay mail to all unknown domains through given IP and port, using queue directory as temporary place to place files.

-d*, *–domain<domain>=<path> generate an SMTP/POP3 virtual maildir domain named "domain" which saves to "path"

-u*, *–username<name>=<password> add a user/password to the last specified domains

-b*, *–bounce_to_postmaster undelivered mails are sent to the postmaster, instead of being rejected.

-p*, *–pop<port> <port> is a number representing which port you want to start the pop3 server on.

-s*, *–smtp<port> <port> is a number representing which port you want to start the smtp server on.

This creates a mail.tap file that can be used by twistd(1)

11.5.8 telnet options

-p*, *–port<port> Run the telnet server on <port>

-u*, *–username<name> set the username to <name>

-w*, *–password<password> set the password to <password>

11.5.9 socks options

-i*, *–interface<interface> Listen on interface <interface>

-p*, *–port<port> Run the SOCKSv4 server on <port>

-l*, *–log<filename> log connection data to <filename>

11.5.10 ftp options

-a*, *–anonymous Allow anonymous logins

-3*, *–thirdparty Allow third party connections

–otp Use one time passwords (OTP)

-p*, *–port<port> Run the FTP server on <port>

-r*, *–root<path> Define the local root of the FTP server

–anonymoususer<username> Define the the name of the anonymous user

11.5.11 manhole options

-p*, *–port<port> Run the manhole server on <port>

-u*, *–user<name> set the username to <name>

-w*, *–password<password> set the password to <password>

11.5.12 words options

-p*, *–port<port> Run the Words server on <port>

-i*, *–irc<port> Run IRC server on port <port>

-w*, *–web<port> Run web server on port <port>

11.5.13 AUTHOR

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

11.5.14 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

11.5.15 COPYRIGHT

11.5.16 SEE ALSO

twistd(1)

11.6 IM.1

11.6.1 NAME

t-im - run Instance Messenger, the GTK+ twisted.words client

11.6.2 SYNOPSIS

t-im

11.6.3 DESCRIPTION

Run Instance Messenger, the GTK+ twisted.words client

11.6.4 AUTHOR

Written by Moshe Zadka, based on t-im's code

11.6.5 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

11.6.6 COPYRIGHT

Copyright ©2000 Matthew W. Lefkowitz This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

11.7 TAP2DEB.1

11.7.1 NAME

tap2deb - create Debian packages which wrap .tap files

11.7.2 SYNOPSIS

tap2deb [options]

11.7.3 DESCRIPTION

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

-u*, *–unsigned do not sign the Debian pacakge

***-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(1)* 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 explanantion about this being an automatic package created from tap2deb.

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

-v*, *–version Output version information and exit.

11.7.4 AUTHOR

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

11.7.5 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

11.7.6 COPYRIGHT

11.7.7 SEE ALSO

mktap(1)

11.8 TAPCONVERT.1

11.8.1 NAME

tapconvert - convert Twisted configurations from one format to another

11.8.2 SYNOPSIS

tapconvert -i *input* -o *output* [-f *input-type*] [-t *output-type*] [-d] [-e]
tapconvert –help

11.8.3 DESCRIPTION

The *–help* prints out a usage message to standard output.

***–in*, *-i*<*input file*>** The name of the input configuration.

***–out*, *-o*<*output file*>** The name of the output configuration.

***–typein*, *-f*<*input type*>** The type of the input file. Can be either 'guess', 'python', 'pickle', 'xml', or 'source'. Default is 'guess'.

***–typeout*, *-t*<*output type*>** The type of the output file. Can be either 'pickle', 'xml', or 'source'. Default is 'source'.

–decrypt*, *-d Decrypt input.

–encrypt*, *-e Encrypt output.

11.8.4 AUTHOR

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

11.8.5 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

11.8.6 COPYRIGHT

Copyright ©2000 Matthew W. Lefkowitz This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

11.8.7 SEE ALSO

mktap(1)

11.9 TRIAL.1

11.9.1 NAME

trial - run unit tests

11.9.2 SYNOPSIS

trial [-b] [-v—-o—-j] [–coverage *coverdir*] [-r *reactor*] [-l *logfile*] [-m *module* [-m *module* ...]] [-p *package* [-p *package* ...]] *file—module—package—TestCase—testMethod* ...

trial –help

11.9.3 DESCRIPTION

trial loads and executes a suite of unit tests, obtained from modules and packages listed on the command line. The *–help* option prints out a usage message to standard output.

-s*, *–summary Print out just a machine-parseable summary of the results.

-v*, *–verbose Be more verbose. Without this option, trial prints out a single character for each test. (e.g. An 'F' for a failure, a '.' for a success, a 'S' for skipped test, a 'T' for a todo and '!' for unexpected success). With this option, trial prints a single line for each test. This is especially useful for gauging how long each test takes.

-o*, *–bwverbose Be verbose, but do not attempt to use colors (more log-file friendly)

-j*, *–jelly Report results in a machine-readable jelly stream.

–timing Report results with timing information for each test.

***–tbformat*<format>** Format to display tracebacks with. Valid values are 'plain' and 'emacs', default being 'plain'.

***-m*, *–module*<module>** Module containing test cases.

***–testmodule*<module>** Find the test case for a named file

***-p*, *–package*<package>** Package containing modules that contain test cases. trial loads modules named 'test_' within the given package.

***-l*, *–logfile*<logfile>** Log exceptions (and other things) to the given logfile.

***-r*, *–reactor*<reactor>** Use this reactor for running the tests. The reactor names are the same as those accepted by twistd: c, qt, gtk2, and so on.

***–coverage*<coverdir>** Generate coverage information in the given directory (relative to _trial_temp). Requires Python 2.3.3.

-b*, *–debug Run the tests in the Python debugger. Also does post-mortem debugging on exceptions.

-R*, *–recurse Recursively search the specified packages for test modules.

11.9.4 AUTHOR

Written by Jonathan M. Lange

11.9.5 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

11.9.6 COPYRIGHT

11.10 TWISTD.1

11.10.1 NAME

twistd - run Twisted applications (TACs, TAPs)

11.10.2 SYNOPSIS

twistd [options]

11.10.3 DESCRIPTION

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

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

-q*, *–quiet be a little more quiet

-p*, *–profile Run profiler

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

-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.

–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 .)

-r*, *–reactor<reactor> Choose which ReactorCore event loop to use, such as 'poll' or 'gtk'.

–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)

-x*, *–xml<tax file> Load an Application from the given .tax (XML) file.

-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 setting, if given, overrides *-f*.

-g*, *–plugin<plugin name> Read config.tac from a plugin package, as with *-y*.

–syslog Log to syslog, not to file.

–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.

11.10.4 AUTHOR

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

11.10.5 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

11.10.6 COPYRIGHT

Copyright ©2003 Matthew W. Lefkowitz This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

11.10.7 SEE ALSO

mktap(1)

11.11 WEBSETROOT.1

11.11.1 NAME

websetroot - set the root of a Twisted web server

11.11.2 SYNOPSIS

websetroot {-f *tapfile* — -y *codefile* — -x *XML* — -s *AOT* } {–pickle *pickle* — –script *script* } [-e] [–port *port*]
websetroot –help

11.11.3 DESCRIPTION

The *–help* prints out a usage message to standard output.

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

***-p*, *–port*<port>** The port the web server is running on [default: 80]

***-f*, *–file*<file>** read the given .tap file [default: twistd.tap]

***-y*, *–python*<file>** read an application from within a Python file

***-x*, *–xml*<file>** Read an application from a .tax file (Marmalade format).

***-s*, *–source*<file>** Read an application from a .tas file (AOT format).

***–script*<file>** Read the root resource from the given resource script file

***–pickle*<file>** Read the root resource from the given resource pickle file

11.11.4 AUTHOR

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

11.11.5 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

11.11.6 COPYRIGHT

11.11.7 SEE ALSO

mktap(1)

Chapter 12

Appendix

12.1 The Twisted FAQ

12.1.1 General

What is “Twisted”?

Please see Twisted[1]

Why should I use Twisted?

See The Twisted Advantage[2]

I have a problem “getting” Twisted.

Did you check the HOWTO collection? There are so many documents there that they might overwhelm you... try starting from the index, reading through the overviews and seeing if there seems to be a chapter which explains what you need to. You can try reading the PostScript or PDF formatted books, inside the distribution. And, remember, the source will be with you... always.

Why is Twisted so big?

Twisted is a lot of things, rolled into one big package. We’re not sure if it’ll stay this way, yet, but for now, if you have only specific needs, we recommend grabbing the big Twisted tarball, and if you want, you can run the ’setup.py’ script with a modified config file to generate a package with only certain Twisted sub-packages. Twisted as a whole makes it into many operating system distributions (FreeBSD, Debian and Gentoo, at least) so size shouldn’t be an issue for the end developer or user. In addition, packaging Twisted as a whole makes sure the end users do not have to worry about versioning parts of Twisted and inter-version compatibility.

If you are distributing Twisted to end-users, you can base your distribution on the “Nodocs” packages, which are signficantly smaller.

[1] http://twistedmatrix.com/products/twisted
[2] http://twistedmatrix.com/services/twisted-advantage

But won't Twisted bloat my program, since it's so big?

No. You only need to import the sub-packages which you want to use, meaning only those will be loaded into memory. So if you write a low-level network protocol, you'd only import twisted.internet, leaving out extraneous things like twisted.web, etc. Twisted itself is very careful with internal dependancies, so importing one subpackage is not likely to import the whole twisted package.

12.1.2 Stability

Does the 1.0 release mean that all of Twisted's APIs are stable?

No, only specific parts of Twisted are stable, i.e. we only promise backwards compatibility for some parts of Twisted. While these APIs may be extended, they will not change in ways that break existing code that uses them.

While other parts of Twisted are not stable, we will however do our best to make sure that there is backwards compatibility for these parts as well. In general, the more the module or package are used, and the closer they are to being feature complete, the more we will concentrate on providing backwards compatibility when API changes take place.

Which parts of Twisted are stable?

Only modules explictily marked as such can be considered stable. Semi-stable modules may change, but not in a large way and some sort of backwards-compatibily will probably be provided. If no comment about API stability is present, assume the module is unstable.

In Twisted 1.1, *most of twisted.internet, .cred and .application are completely stable* (excepting of course code marked as deprecated).

But as always, the only accurate way of knowing a module's stability is reading the module's docstrings.

12.1.3 Installation

I run mktap (from site-packages/twisted/scripts/mktap.py) and nothing happens!

Don't run scripts out of `site-packages`. The Windows installer should install executable scripts to someplace like `C:\Python22\scripts\`, *nix installers put them in `$PREFIX/bin`, which should be in your $PATH.

Why do the Debian packages for Alphas and Release Candidates have weird versions containing old version numbers?

An example: 1.0.6+1.0.7rc1-1

In Debian versioning, 1.0.7rc1 is *greater than* 1.0.7. This means that if you install a package with Version: 1.0.7rc1, and then that package gets a new version 1.0.7, apt will not upgrade it for you, because 1.0.7 looks like an older version. So, we prefix the previous version to the actual version. 1.0.6+1.0.7rc1 is *less than* 1.0.7.

12.1.4 Core Twisted

How can I access self.factory from my Protocol's __init__?

You can't. A Protocol doesn't have a Factory when it is created. Instead, you should probably be doing that in your Protocol's `connectionMade` method.

Similarly you shouldn't be doing "real" work, like connecting to databases, in a Factory's `__init__` either. Instead, do that in `startFactory`.

See *Writing Servers* (page 98) and *Writing Clients* (page 103) for more details.

Where can I find out how to write Twisted servers?

Try *Writing Servers* (page 98).

When I try to install my reactor, I get errors about a reactor already being installed. What gives?

Here's the rule - installing a reactor should always be the *first* thing you do, and I do mean first. Importing other stuff before you install the reactor can break your code.

Tkinter and wxPython support, as they do not install a new reactor, can be done at any point, IIRC.

twistd won't load my .tap file! What's this Ephemeral nonsense?

When the pickled application state cannot be loaded for some reason, it is common to get a rather opaque error like so:

```
% twistd -f test2.tap

Failed to load application: global name 'initRun' is not defined
```

The rest of the error will try to explain how to solve this problem, but a short comment first: this error is indeed terse – but there is probably more data available elsewhere – namely, the `twistd.log` file. Open it up to see the full exception.

The error might also look like this:

```
Failed to load application: <twisted.persisted.styles.Ephemeral instance at
0x82450a4> is not safe for unpickling
```

To load a `.tap` file, as with any unpickling operation, all the classes used by all the objects inside it must be accessible at the time of the reload. This may require the PYTHONPATH variable to have the same directories as were available when the application was first pickled.

A common problem occurs in single-file programs which define a few classes, then create instances of those classes for use in a server of some sort. If the class is used directly, the name of the class will be recorded in the `.tap` file as something like `__main__.MyProtocol`. When the application is reloaded, it will look for the class definition in `__main__`, which probably won't have it. The unpickling routines need to know the module name, and therefore the source file, from which the class definition can be loaded.

The way to fix this is to import the class from the same source file that defines it: if your source file is called `myprogram.py` and defines a class called `MyProtocol`, you will need to do a `from myprogram import MyProtocol` before (and in the same namespace as) the code that references the MyProtocol class. This makes it important to write the module cleanly: doing an `import myprogram` should only define classes, and should not cause any other subroutines to get run. All the code that builds the Application and saves it out to a `.tap` file must be inside an `if __name__ == '__main__'` clause to make sure it is not run twice (or more).

When you import the class from the module using an "external" name, that name will be recorded in the pickled `.tap` file. When the `.tap` is reloaded by `twistd`, it will look for `myprogram.py` to provide the definition of `MyProtocol`.

Here is a short example of this technique:

```
# file dummy.py
from twisted.internet import protocol
class Dummy(protocol.Protocol): pass
if __name__ == '__main__':
    from twisted.application import service, internet
    a = service.Application("dummy")
    import dummy
    f = protocol.Factory()
    f.protocol = dummy.Dummy # Note! Not "Dummy"
    internet.TCPServer(2000, f).setServiceParent(service.IServiceCollect(a))
    a.save()
```

I get "Interrupted system call" errors when I use os.popen2. How do I read results from a sub-process in Twisted?

You should be using `reactor.spawnProcess` (see `interfaces.IReactorProcess.spawnProcess`). There's also a convenience function, `getProcessOutput`, in `twisted.internet.utils`.

Why don't my spawnProcess programs see my environment variables?

`spawnProcess` defaults to clearing the environment of child processes as a security feature. You can either provide a dictionary with exactly the name-value pairs you want the child to use, or you can simply pass in `os.environ` to inherit the complete environment.

My Deferred or DeferredList never fires, so my program just mysteriously hangs! What's wrong?

It really depends on what your program is doing, but the most common cause is this: it *is* firing – but it's an error, not a success, and you have forgotten to add an *errback* (page 381), so nothing happens. Always add errbacks!

The reason this happens is that unhandled errors in Deferreds get printed when the Deferred is garbage collected. Make sure your Deferred is garbage collected by deleting all references to it when you are done with it, e.g. after `callback()` is called.

My exceptions and tracebacks aren't getting printed!

See previous question.

How do I use Deferreds to make my blocking code non-blocking?

You don't. Deferreds don't magically turn a blocking function call into a non-blocking one. A Deferred is just a simple object that represents a *deferred result*, with methods to allow convenient adding of callbacks. (This is a common misunderstanding; suggestions on how to make this clearer in the *Deferred Execution* (page 120) howto are welcome!)

If you have blocking code that you want to use non-blockingly in Twisted, either rewrite it to be non-blocking, or run it in a thread. There is a convenience function, `deferToThread`, to help you with the threaded approach – but be sure to read *Using Threads in Twisted* (page 134).

I get "exceptions.ValueError: signal only works in main thread" when I try to run my Twisted program! What's wrong?

The default reactor, by default, will install signal handlers to catch events like Ctrl-C, SIGTERM, and so on. However, you can't install signal handlers from non-main threads in Python, which means that `reactor.run()` will cause an error. Pass the `installSignalHandlers=0` keyword argument to `reactor.run` (or `Application.run`) to work around this.

I'm trying to stop my program with sys.exit(), but Twisted seems to catch it! How do I exit my program?

Use `reactor.stop()` instead. This will cleanly shutdown the reactor.

How do I find out the IP address of the other end of my connection?

The `.transport` object (which implements the `ITransport` interface) offers a pair of methods named `getPeer` and `getHost`. `getPeer` will give you a tuple that describes the address of the system at the other end of the connection. For example:

```
class MyProtocol(protocol.Protocol):
    def connectionMade(self):
        print "connection from", self.transport.getPeer()
```

12.1.5 Web

Is the Twisted web server a toy?

No. It is a production grade server. It is running continously on several sites and has been proven quite stable. The server can take loads of up to 3000 users at a time and still keep churning several million requests a day, even on low end hardware. It can serve static files or dynamically rendered pages.

But can Twisted Web do PHP?

Yes. It works out-of-the-box, so long as you've got the standalone php interpreter installed.

And can Twisted Web do virtual hosting?

Can it ever!

You can decide to go with one big process for all of them, a front server and a seperate server for each virtual host (for example, for permission reasons), and you can even mix-and-match between Apache and Twisted (for example, put Apache in the front and have Twisted handle some subset of the virtual host).

How do I use twisted.web to do complex things?

See *the Twisted.Web Howto* (page 247).

I've been using Woven since before it was called Woven. I just upgraded and now I'm getting a confusing traceback talking about INodeMutator. What gives?

You probably have code that's survived the upgrade from PyXML's `minidom` to Twisted's `microdom`. Try deleting any `.pxp` files that you have lying around and the error will probably go away.

My Woven pages are sent to the browser with a trailing slash appended to the URL, which breaks all of the relative links. How do I get rid of the trailing slash?

If you are subclassing `Page`, you can add a class attribute `addSlash = 0`, like this:

```
class Foo(page.Page):
        addSlash = 0
```

If you are still subclassing `Controller`, you can put the `addSlash = 0` there. Consider subclassing `Page` instead, as having a `Model`, `View`, `Controller` triad as the base of a `Page` will be deprecated in the near future.

If you're just using the generic `Page` instance, you can set it after creation like this:

```
resource = page.Page("foo")
resource.addSlash = 0
```

The default behavior of Woven is now to automatically add a slash because it makes creating relative links far easier, ironically ;-)

Argh! When using Woven, my newlines get mangled inside a <pre>

Use the `RawText` view.

When trying to use Guard, I get infinite redirects to URLs with long hexadecimal numbers. What's the deal?

Are you using an RPY? Add a single line containing `cache()` as the first line of your RPY.

An RPY file is executed every time it is accessed, so the call to `guard.SessionWrapper` is executed once per request, making a new SessionWrapper for each request. There should be only one SessionWrapper object for your application. Caching the RPY file enforces this.

12.1.6 Perspective Broker

How can I get the reference to a client from a Perspective?

Firstly, the client must send a reference when it connects to the perspective broker. This can be done by passing the reference as a parameter to `pb.connect`.

At the server end, you must override the `Perspective.attach`, which is called when a client attaches to a perspective. The first argument of this method is a remote reference to the client object that was passed to `pb.connect`.

Note that a single perspective can have many attached clients. For further information, see *Managing Clients of Perspectives* (page 243) HOWTO and the `twisted.spread.pb` API docs.

12.1.7 Requests and Contributing

Twisted is cool, but I need to add more functionality.

Great! Read our the docs, and if you're feeling generous, contribute patches.

I have a patch. How do I maximize the chances the Twisted developers will include it?

Use unified diff. Either use `svn diff` or, better yet, make a clean checkout and use `diff -urN` between them. Make sure your patch applies cleanly. In your post to the mailing list, make sure it is inlined and without any word wrapping.

And to whom do I send it?

Add it to the bug tracker[3], and if it's an urgent or important issue you may want to tell the mailing list[4]. about the issue you added

My company would love to use Twisted, but it's missing feature X, can you implement it?

You have 3 options:

- Pay one of the Twisted developers to implement the feature.
- Implement the feature yourself.
- Add a feature request to our bug tracker. We will try to implement the feature, but there are no guarantees when and if this will happen.

12.1.8 Documentation

Twisted really needs documentation for X, Y or Z - how come it's not documented?.

We are doing the best we can, and there is documentation in progress for many parts of Twisted. There is a limit to how much we can do in our free time. See also the answer to the next question.

Wow the Twisted documentation is nice! I want my docs to look like that too!

Now you can, with `twisted.lore`. See the manual page for `lore`. For source format documentation, see *the documentation standard description* (page 341). For a more comprehensive explanation, see *the Lore HOWTO* (page 318).

[3] http://twistedmatrix.com/bugs/
[4] http://twistedmatrix.com/cgi-bin/mailman/listinfo/twisted-python

12.1.9 Communicating with us

There's a bug in Twisted. Where do I report it?

Unless it is a show-stopper bug, we usually won't fix it if it's already fixed in Subversion[5], so you would do well to look there. Then send any pertinent information about the bug (hopefully as much information needed to reproduce it: OS, Subversion versions of any important files, Python version, code you wrote or things you did to trigger the bug, etc.) to the mailing list[6]. If no one answers immediately, you should add it to the bug tracker[7].

Where do I go for help?

Ask for help where the Twisted team hangs out[8]

How do I e-mail a Twisted developer?

First, note that in many cases this is the wrong thing to do: if you have a question about a part of Twisted, it's usually better to e-mail the mailing list. However, the preferred e-mail addresses for all Twisted developers are listed in the file "CREDITS" in the Subversion repository.

12.2 Twisted Glossary

Absolute submodel paths The full path to a *Model* (page 382) object, starting at the root. For example, `/foo/bar/baz`

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 143).

`Application` A `twisted.application.service.Application`. There are HOWTOs on *creating and manipulating* (page 141) them as a system-administrator, as well as *using* (page 154) them in your code.

`Authorizer` (Deprecated; this is part old cred) An object responsible for managing *Identities* (page 381). See `twisted.cred.authorizer`.

Avatar (from *Twisted Cred* (page 381)) business logic for specific user. For example, in *PB* (page 382) these are perspectives, in pop3 these are mailboxes, and so on.

`Banana` The low-level data marshalling layer of *Twisted Spread* (page 383). See `twisted.spread.banana`.

`Broker` A `twisted.spread.pb.Broker`, the object request broker for *Twisted Spread* (page 383).

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.

[5] http://twistedmatrix.com/developers/cvs
[6] http://twistedmatrix.com/cgi-bin/mailman/listinfo/twisted-python
[7] http://twistedmatrix.com/bugs/
[8] http://twistedmatrix.com/services/online-help

COIL "COnfiguration ILlumination". It is a (stagnant and incomplete) end-user interface for configuring Twisted applications. See `twisted.coil`.

component A special kind of (persistent) `Adapter` that works with a `twisted.python.components.Componentized`. See also *Interfaces and Adapters* (page 143).

`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 143).

`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 103).

Consumer An object that consumes data from a *Producer* (page 382). See `twisted.internet.interfaces.IConsumer`.

controller (In *Woven* (page 384)) an object which accepts input from the user in the form of mouse clicks, keypresses, and web form submissions, and updates the *Model* (page 382) component.

Cred Twisted's authentication API, `twisted.cred`. See *Introduction to Twisted Cred* (page 149) and *Twisted Cred usage* (page 227).

credentials A username/password, public key, or some other information used for authentication.

credential checker Where authentication actually happens. See `ICredentialChecker`.

CVSToys A nifty set of tools for CVS, available at http://twistedmatrix.com/users/acapnotic/wares/code/CVSToys/.

`Deferred` A instance of `twisted.internet.defer.Deferred`, an abstraction for handling chains of callbacks and error handlers ("errbacks"). See the *Deferring Execution* (page 120) HOWTO.

Enterprise Twisted's RDBMS support. It contains `twisted.enterprise.adbapi` for asynchronous access to any standard DB-API 2.0 module, and `twisted.enterprise.row`, a "*Relational Object Wrapper* (page 383)". See *Introduction to Twisted Enterprise* (page 186) and *Twisted Enterprise Row Objects* (page 189) 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* (page 383) instances for incoming or outgoing connections. See *Writing Servers* (page 98) and *Writing Clients* (page 103).

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

`Identity` (Deprecated). A *Cred* (this page) object that represents a single user with a username and a password of some sort.

im, t-im Abbreviation of "(Twisted) *Instance Messenger* (page 382)".

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* (page 382) with *Twisted Words* (page 384). See `twisted.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 143).

Jelly The serialization layer for *Twisted Spread* (page 383), 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 380), in most cases). See `twisted.spread.jelly`.

Lore `twisted.lore` is Twisted's documentation system. The source format is a subset of XHTML, and output formats include HTML and LaTeX. See *lore(1)* (page 356) and the *Twisted Documentation Standard* (page 341).

Manhole A debugging/administration interface to a Twisted application.

Marmalade An XML-based serialisation module. See `twisted.persisted.marmalade`.

Microdom A partial DOM implementation using *SUX* (page 383). It is simple and pythonic, rather than strictly standards-compliant. See `twisted.web.microdom`.

model An object that contains data and business logic for manipulating this data.

model stack A stack of *Model* (this page) instances which keeps track of the Model that is currently in scope during the *Woven* (page 384) Page rendering process.

Names Twisted's DNS server, found in `twisted.names`.

overrides (In *Woven* (page 384)) A way to add data to cached data. Overrides are not replaced when base data is updated.

pattern A node in a *Woven* (page 384) HTML template whose sole purpose is to be copied and filled with data by a View component.

relative submodel path A partial path to a *Model* (this page) object, relative to the top of the Model stack.

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

`Perspective` A *Cred* (page 381) object; an *Identity* (page 381)'s "perspective" (or "view") onto a Service. There may be many Perspectives associated with an Identity, and an Identity may have multiple Perspectives onto the same *Service* (page 383).

Perspective Broker The high-level object layer of Twisted *Spread* (page 383), implementing semantics for method calling and object copying, caching, and referencing. See `twisted.spread.pb`.

Portal Glues *credential checkers* (page 381) and *realm* (page 383)s together.

Producer An object that generates data a chunk at a time, usually to be processed by a *Consumer* (page 381). 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 98) and *Writing Clients* (page 103).

PSU There is no PSU.

Reactor The core event-loop of a Twisted application. See *Reactor Basics* (page 97).

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

realm (in *Twisted Cred* (page 381)) stores *avatars* (page 380) and perhaps general business logic. See `IRealm`.

`Resource` A `twisted.web.resource.Resource`, which are served by Twisted Web. Resources can be as simple as a static file on disk, or they can have dynamically generated content.

ROW *R*elational *O*bject *W*rapper, an object-oriented interface to a relational database. See *Twisted Enterprise Row Objects* (page 189).

Service A `twisted.application.service.Service`. See *Application howto* (page 154) for a description of how they relate to *Applications* (page 380).

Spread Twisted Spread[9] is Twisted's remote-object suite. It consists of three layers: *Perspective Broker* (page 382), *Jelly* (page 382) and *Banana.* (page 380) See *Writing Applications with Perspective Broker* (page 15).

Sturdy A persistent reference manager for *PB* (page 382). See `twisted.spread.sturdy`.

submodel paths A path to a *Model* (page 382) object. A way of referring to a piece of data in a *Woven* (page 384) template that allows Python to locate the data in a tree of python objects.

SUX *S*mall *U*ncomplicated *X*ML, Twisted's simple XML parser written in pure Python. See `twisted.protocols.sux`.

TAP *T*wisted *A*pplication *P*ickle, or simply just a *T*wisted *AP*plication. A serialised application that created with `mktap` and runnable by `twistd`. See *Using the Utilities* (page 141).

Tendril A bridge between *Twisted Words* (page 384) and IRC. See `twisted.words.tendril`.

Trial `twisted.trial`, Twisted's unit-testing framework, modelled after pyunit[10]. See also Twisted's *Test Standard* (page 344).

Twisted Matrix Laboratories The team behind Twisted. http://twistedmatrix.com/.

Twisted Reality In days of old, the Twisted Reality[11] 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 since been broken off into a separate 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 181).

---

[9] http://twistedmatrix.com/products/spread
[10] http://pyunit.sourceforge.net/
[11] http://twistedmatrix.com/products/reality

`wcfactory` A *factory* (page 381) method for producing *Controller* (page 381) objects when a `controller=` directive is encountered in a *Woven* (page 384) Template.

`wchild` In *Woven* (page 384), a *factory* (page 381) method for producing objects which represent URL segments below the current object.

widgets In *Woven* (this page), a `View` subclass which specializes in rendering a fragment of the DOM tree.

`wmfactory` A *factory* (page 381) method for producing *Model* (page 382) objects when a `model=` directive is encountered in a *Woven* (this page) Template.

Words Twisted Words is a multi-protocol chat server that uses the *Perspective Broker* (page 382) protocol as its native communication style. See `twisted.words`.

Woven *W*eb *O*bject *V*isualization *En*vironment. A web templating system based on XML and the Model-View-Controller design pattern. See *Developing Componentized Applications using Woven* (page 267).

`wvfactory` A *factory* (page 381) method for producing View objects when a `view=` directive is encoutered in a *Woven* (this page)Template.

Zoot Twisted's Gnutella implementation (currently very incomplete). See `twisted.zoot`.

12.3 Banana Protocol Specifications

12.3.1 Introduction

Banana is an efficient, extendable protocol for sending and receiving s-expressions. A s-expression in this context is a list composed of byte strings, integers, large integers, floats and/or s-expressions.

12.3.2 Banana Encodings

The banana protocol is a stream of data composed of elements. Each element has the following general structure - first, the length of element encoded in base-128, least signficant bit first. For example length 4674 will be sent as `0x42 0x24`. For certain element types the length will be omitted (e.g. float) or have a different meaning (it is the actual value of integer elements).

Following the length is a delimiter byte, which tells us what kind of element this is. Depending on the element type, there will then follow the number of bytes specified in the length. The byte's high-bit will always be set, so that we can differentiate between it and the length (since the length bytes use 128-base, their high bit will never be set).

12.3.3 Element Types

Given a series of bytes that gave us length N, these are the different delimiter bytes:

List – 0x80 The following bytes are a list of N elements. Lists may be nested, and a child list counts as only one element to its parent (regardless of how many elements the child list contains).

Integer – 0x81 The value of this element is the positive integer N. Following bytes are not part of this element. Integers can have values of 0 <= N <= 2147483647.

String – 0x82 The following N bytes are a string element.

Negative Integer – 0x83 The value of this element is the integer N * -1, i.e. -N. Following bytes are not part of this element. Negative integers can have values of 0 >= -N >= -2147483648.

Float - 0x84 The next 8 bytes are the float encoded in IEEE 754 floating-point "double format" bit layout. No length bytes should have been defined.

Large Integer – 0x85 The value of this element is the positive large integer N. Following bytes are not part of this element. Large integers have no size limitation.

Large Negative Integer – 0x86 The value of this element is the negative large integer -N. Following bytes are not part of this element. Large integers have no size limitation.

Large integers are intended for arbitary length integers. Regular integers types (positive and negative) are limited to 32-bit values.

Examples

Here are some examples of elements and their encodings - the type bytes are marked in bold:

`1` `0x01 0x81`

`-1` `0x01 0x83`

`1.5` `0x84 0x3f 0xf8 0x00 0x00 0x00 0x00 0x00 0x00`

`"hello"` `0x05 0x82 0x68 0x65 0x6c 0x6c 0x6f`

`[]` `0x00 0x80`

`[1, 23]` `0x02 0x80 0x01 0x81 0x17 0x81`

`123456789123456789` `0x15 0x3e 0x41 0x66 0x3a 0x69 0x26 0x5b 0x01 0x85`

`[1, ["hello"]]` `0x02 0x80 0x01 0x81 0x01 0x80 0x05 0x82 0x68 0x65 0x6c 0x6c 0x6f`

12.3.4 Profiles

The Banana protocol is extendable. Therefore, it supports the concept of profiles. Profiles allow developers to extend the banana protocol, adding new element types, while still keeping backwards compatability with implementations that don't support the extensions. The profile used in each session is determined at the handshake stage (see below.)

A profile is specified by a unique string. This specification defines two profiles - `"none"` and `"pb"`. The `"none"` profile is the standard profile that should be supported by all Banana implementations. Additional profiles may be added in the future.

The "none" Profile

The `"none"` profile is identical to the delimiter types listed above. It is highly recommended that all Banana clients and servers support the `"none"` profile.

The "pb" Profile

The `"pb"` profile is intended for use with the Perspective Broker protocol, that runs on top of Banana. Basically, it converts commonly used PB strings into shorter versions, thus minimizing bandwidth usage. It does this by adding an additional delimiter byte, 0x87. This byte should not be prefixed by a length. It should be followed by a single byte, which tells us to which string element to convert it:

0x01 'None'

0x02 'class'

0x03 'dereference'

0x04 'reference'

0x05 'dictionary'

0x06 'function'

0x07 'instance'

0x08 'list'

0x09 'module'

0x0a 'persistent'

0x0b 'tuple'

0x0c 'unpersistable'

0x0d 'copy'

0x0e 'cache'

0x0f 'cached'

0x10 'remote'

0x11 'local'

0x12 'lcache'

0x13 'version'

0x14 'login'

0x15 'password'

0x16 'challenge'

0x17 'logged_in'

0x18 'not_logged_in'

0x19 'cachemessage'

0x1a 'message'

0x1b 'answer'

0x1c 'error'

0x1d 'decref'

0x1e 'decache'

0x1f 'uncache'

12.3.5 Protocol Handshake and Behaviour

The initiating side of the connection will be referred to as "client", and the other side as "server".

Upon connection, the server will send the client a list of string elements, signifying the profiles it supports. It is recommended that `"none"` be included in this list. The client then sends the server a string from this list, telling the server which profile it wants to use. At this point the whole session will use this profile.

Once a profile has been established, the two sides may start exchanging elements. There is no limitation on order or dependencies of messages. Any such limitation (e.g. "server can only send an element to client in response to a request from client") is application specific.

Upon receiving illegal messages, failed handshakes, etc., a Banana client or server should close its connection.