The Twisted Documentation

The Twisted Development Team

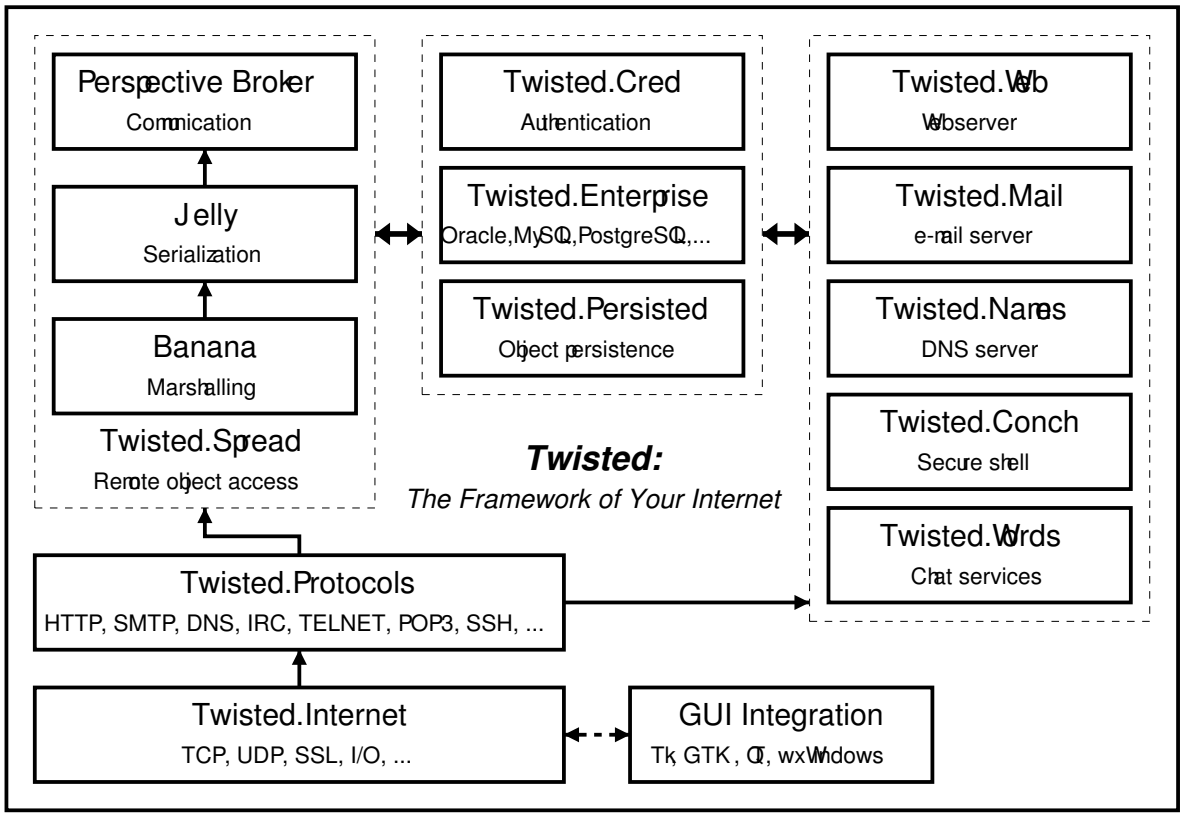
January 29, 2003

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.

Twisted should be a collaboration application. The next major phase of development will also involve lots of chat, mail, and news functionality, both in clients and in servers.

Finally, Twisted should be a personal information space as well as a shared one. Twisted should unify all your messages and contacts for you across multiple machines and in multiple environments, through multiple modes of access, while also being industrial-strength enough to run the back end of an online sales service with millions of users.

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.

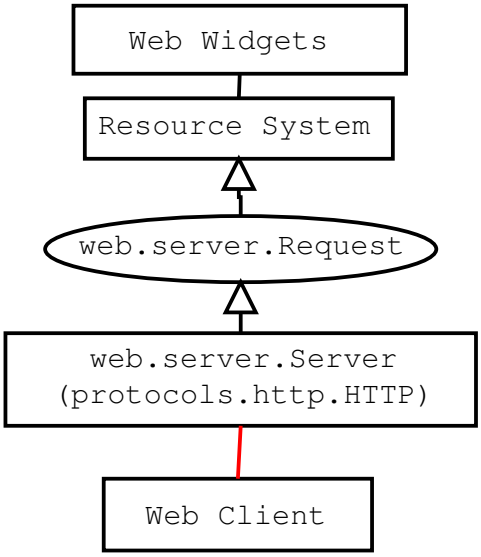
The `app.Application` class allows for a similar API to the reactor, which is automatically persistent. Applications usually persist and resurrect automatically, depending on the usage. See the Application documentation (page 35) for more information.

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. The most useful for web application designers is Web Widgets (page 142), a high-level class-and-template oriented system. There is also the Resource system, which Web Widgets is built on.

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 Widgets

Web Widgets are an added layer of abstraction of Resources – they're much nicer for most sorts of web applications. For more information on Widgets, see Introducing Web Widgets (page 142).

1.5 Overview of Twisted Spread

Perspective Broker (affectionately known as "PB") is an asynchronous, symmetric[1], network protocol for secure, remote method calls. 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.

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.

1.6 Introduction to Twisted Enterprise

1.6.1 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.

1.6.2 What you should already know

- Python :-)
- How to write a simple Twisted Server (see this tutorial (page 58) to learn how)
- Familiarity with using database interfaces (see the documentation for DBAPI 2.0[2] or this article[3] by Andrew Kuchling)

1.6.3 Quick Overview

Twisted is an asynchronous framework. This means standard database modules cannot be used directly, as they typically work something like:

[1] 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.

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

[3] http://www.amk.ca/python/writing/DB-API.html

```
# 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[4]-compliant module. It is currently best tested with the pyPgSQL[5] module for PostgreSQL[6].

`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.

1.6.4 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.
- You may also control the size of the connection pool with the keyword parameters `cp_min` and `cp_max`. The default minimum and maximum values are 3 and 5.

So, now you need to be able to dispatch queries to your ConnectionPool. We do this by subclassing `adbapi.Augmentation`. Here's an example:

```
class AgeDatabase(adbapi.Augmentation):
    """A simple example that can retrieve an age from the database"""
    def getAge(self, name):
        # Define the query
```

[4] http://www.python.org/topics/database/DatabaseAPI-2.0.html
[5] http://pypgsql.sourceforge.net/
[6] http://www.postgresql.org/

```
        sql = """SELECT Age FROM People WHERE name = ?"""
        # Run the query, and return a Deferred to the caller to add
        # callbacks to.
        return self.runQuery(sql, name)


def gotAge(resultlist, name):
    """Callback for handling the result of the query"""
    age = resultlist[0][0]          # First field of first record
    print "%s is %d years old" % (name, age)

db = AgeDatabase(dbpool)

# These will *not* block.  Hooray!
db.getAge("Andrew").addCallbacks(gotAge, db.operationError,
                                 callbackArgs=("Andrew",))
db.getAge("Glyph").addCallbacks(gotAge, db.operationError,
                                callbackArgs=("Glyph",))

# Of course, nothing will happen until the reactor is started
from twisted.internet import reactor
reactor.run()
```

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 72).

Also worth noting is that this example 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, ...)`.

1.6.5 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.

1.7 Why and How to use Twisted.Cred

1.7.1 Authentication and Account Management in Twisted

(This document is a work in progress. Later it will include some examples but for now a brief explanation is better than nothing!)

Twisted unifies authentication and account management of multiple services in the Twisted.Cred package. Although this authentication model was originally designed to integrate services in the Perspective Broker (page 13) remote method invocation protocol, it is useful in many kinds of servers, and work is underway to move all systems that require log-in in Twisted to use twisted.cred.

In order to use twisted.cred, your code has to be structured around a subclass of `Service`. A service is a particular unit of functionality which has a way to request `Perspective` objects. You will probably have to subclass both of these.

In order to simplify integration of services that come from lots of different places, Twisted.Cred presents user-account related information in two different ways. Application-independent user information, such as passwords, public keys, and other things related to the existence and authentication of a particular person should reside in an `Identity`. Information related to a particular service, such as e-mail messages, high scores, or to-do lists should be represented by a `Perspective`.

In support of these two basic abstractions is the `Authorizer`. An authorizer serves primarily as the storage mechanism for a collection of identities. Its usage varies depending on whether the services it is providing authentication for can support multiple services on one port. `Authorizer` is an abstract class, but you don't need to implement your own; the simplest authorizer to get started with is `DefaultAuthorizer`.

At this point, there are basically 2 ways that an authorizer can be used. It is either the root of a PB object hierarchy, or simply the authorizer for some number of non-PB services.

Setting Up a Service

```
from twisted.internet.app import MultiService
# A service which collects other services.
from twisted.cred.authorizer import DefaultAuthorizer
# A simple in-memory Authorizer implementation.
from my.service import MyService, OtherService
# Two sample user-written services.

multiserv = MultiService("pb")
# multiservice named "pb" to hold other services
auth = DefaultAuthorizer()
auth.setApplication(multiserv)
# the authorizer for both of our other services
myserv = MyService("my service", multiserv, auth)
otherserv = OtherService("another service", multiserv, auth)
# create both of our services pointing to their authorizer

from twisted.internet import reactor
from twisted.spread import pb
reactor.listenTCP(pb.portno, pb.AuthRoot(auth))
# If the services are all pb.Service subclasses, we can connect them to a
# network like this.  It will look up services through the serviceCollection
# passed to the Authorizer; which in this case was a MultiService but could
# also be an Application.
```

1.8 Overview of Twisted IM

Twisted IM (Instance Messenger) is a multi-protocol chat framework, based on the Twisted framework we've all come to know and love. It's fairly simple and extensible in two directions - it's pretty easy to add new protocols, and it's

also quite easy to add new front-ends.

1.8.1 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 `NotImplementedError`s. In your subclass, override the method with a new method whose Class parameter defaults to your own implementation of `Conversation`, that simply calls the parent class' implementation.

Conversation and GroupConversation

These classes are where your interface meets the chat protocol. Chat protocols get a message, find the appropriate `Conversation` or `GroupConversation` object, and call its methods when various interesting things happen.

Override whatever methods you want to get the information you want to display. You must override the `hide` and `show` methods, however - they are called frequently and the default implementation raises `NotImplementedError`.

Accounts

An account is an instance of a subclass of `basesupport.AbstractAccount`. For more details and sample code, see the various `*support` files in `twisted.im`.

Chapter 2

The Basics

2.1 Installing Twisted

2.1.1 Installation

If you are on Windows, you may want to skip this and simply get the Windows Installer version of Twisted from the download page[1].

If you are on Debian. you may want to use the Debian packages. The last stable release of Twisted is at "deb http://twistedmatrix.com/users/moshez/apt ./", and the last prerelease of Twisted is at "deb http://twistedmatrix.com/users/moshez/snapshot ./"

To install Twisted, just make sure the `Twisted-$VERSION/` directory is in the `PYTHONPATH` environment variable. For example, if you extracted Twisted-1.0.2.tar.gz to /home/bob/, then you would have something like:

```
export PYTHONPATH=$PYTHONPATH:/home/bob/Twisted-1.0.2/
```

in your `~/.bash_profile`, `~/.zshrc`, `~/.cshrc`, etc. If you use Windows NT, 2000, or XP, then set your environment variables by right-clicking on My Computer and selecting Properties, then the Advanced tab, and click on the "Environment Variables" button. If you use some other version of windows, you'll need to set the variable at a command prompt, or in autoexec.bat, with the 'set' command.

If you'd like to install Twisted system-wide on your machine and into the default `PYTHONPATH`, you can use `setup.py` to do so:

```
# python ./setup.py install
```

Be sure to run `setup.py` with appropriate privileges (root under Unix).

2.1.2 Optional Compilation

There are a couple of small optional alternative implementations of pieces of Twisted that are in C for increased performance. If you don't run the installer, and you need these modules, you'll need to perform a couple of extra steps:

[1] http://www.twistedmatrix.com/products/download

```
$ python ./setup.py build_ext
```

This will (eventually) generate some shared libraries (cBanana.so, cReactor.so) within a directory tree called 'build' under the Twisted directory.

If you don't go on to install the build results into a directory on the `$PYTHONPATH`, then you will need to create a couple of symlinks:

```
$ cd twisted/spread
$ ln -s ../../build/lib.linux-i686-2.1/twisted/spread/cBanana.so cBanana.so
$ cd ../internet
$ ln -s ../../build/lib.linux-i686-2.1/twisted/internet/cReactor.so cReactor.so
```

The exact details of the symlinks may vary based on your system.

2.1.3 Running Tests

See our unit tests run in a lovely Tkinter GUI, proving that the software is BugFree(TM):

```
% admin/runtests
```

(From the directory where Twisted was originally untarred/unzipped to.)

Some of these tests will fail if you don't have the Crypto packages installed on your system.

2.2 The Basics

2.2.1 Application

Twisted programs usually work with `twisted.internet.app.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.

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 `Application`s.

`Application`s 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.

2.2.2 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.

2.2.3 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.

2.2.4 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 and calling its `.run` method 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 reactor (page 84)), 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 `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.

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

2.2.5 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[2].

2.3 Configuring and Using the Twisted.Web Server

2.3.1 Installation

To install the Twisted.Web server, you'll need to have installed Twisted (page 19).

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
```

2.3.2 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.

[2] http://twistedmatrix.com/users/moshez/debian

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(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(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
```

DOM Templates

The DOM Templates system is a system for handling templated content. See its documentation (page 133) for more details.

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:

```
% 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.internet import app
from twisted.web import proxy, server, vhost
vhostName = 'example.com'
reverseProxy = proxy.ReverseProxyResource('internal', 8538,
                                          '/vhost.rpy/http/'+vhostName+'/')
root = vhost.NamedVirtualHost()
root.addHost(vhostName, reverseProxy)
site = server.Site(root)
application = app.Application('web-proxy')
application.listenTCP(80, site)
```

2.3.3 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.

2.4 Debugging with Manhole

2.4.1 Creating the Manhole Service

In order to create a manhole server, use a command like `mktap manhole -u [username] -w [password]`. If you've already got a "TAP" for a server, you can use the argument `--append [tapname]` to `mktap` to add a manhole service to that "TAP".

2.4.2 Using the Manhole PB Client

The second service offered by twisted.manhole is a Perspective Broker -based server. This gives the client a remote reference to a twisted.manhole.service.Service object, which offers remotely-callable methods to evaluate Python code. With the rich remote-method-invocation facilities provided by PB, however, much more is possible: the client can ask to "watch" certain objects, and then will receive messages every time that python object is changed. (this takes advantage of some twisted.python code that "hooks" some functions, like .setattr). These features are described in detail below.

With this in place and running, you're ready to connect with the manhole client. This is a Gtk+-based GUI application named `manhole` that gets installed along with the rest of twisted. Execute the command `manhole` to start the client, and it will bring up a dialog that asks for hostname, port number, Service name, username, and password (and also "Perspective" but don't worry about that for now). Use the default host/port of localhost/8787 to indicate where the `twisted.manhole` service is listening, and use boss/sekrit for the username and password. Use the default Service name "twisted.manhole", and leave the Perspective blank.

Click the "Log In" button to establish the connection, and you will be greeted with a short message in a window with an output area in the top, and an input area at the bottom. This is just like the python interpreter accessed through the telnet shell, but with a different GUI. You can type arbitrary python code into the input area and get the results in the output area. Note that multi-line sequences are all sent together, so if you define a function (or anything else that uses indentation to tell the interpreter that you aren't finished yet), you'll need to type one additional Return to tell the client to send off the code.

At this point, you can get access to the main `Application` object just like you did before with the telnet-based shell. You can use that to obtain the `Service` objects inside it, or references to the `Factory` objects that are listening on TCP or UDP ports, by doing:

```
from twisted.internet import app
a = app.theApplication
service = a.getServiceNamed("manhole")
(port, factory, backlog, interface) = a.tcpPorts[0]
```

After that, you can do anything you want with those objects.

2.4.3 Special Commands

There are a few special commands so far that make debugging Twisted objects really nice. These are `/browse` and `/watch`. You can `/browse` any type of object, and it will give you some nice information about that object in the "Spelunking" window that pops up when `manhole` establishes a connection to the `manhole` Service. `/watch`-ing an object adds hooks to the object, allowing you to watch modifications to it in real time. Try the following in the `manhole` window and watch what happens in the "Spelunking" box (word wrapped for clarity):

```
/browse ["hello", "there"]
 <ObjectLink of ["hello", "there"] type list>:
  ['hello',
   'there',]


class A:
    def foo(self):
        self.x = 1

x = A()
/browse x
<ObjectLink of x type instance>:
  {members: {}
   class: 'A'
   methods: {}}


/watch x
  <ObjectLink of x type instance>:
  {members: {}
   class: 'A'
   methods: {}}


x.foo()

<ObjectLink of x type instance>:
  {members: {x: 1}
   class: 'twisted.python.explorer.WatchingA8195574'
   methods:
     {foo:
        <ObjectLink of x.foo type instance_method>:
          {class: 'twisted.python.explorer.WatchingA8195574'
           self: '<twisted.python.explorer.WatchingA8195574 instance at
                 0x8195574>'
           doc:
             Pretend to be the method I replaced, and ring the bell.
```

```
            line: 651
            signature:
              [{name: instance},
               {name: a
                list: 1},
               {name: kw
                keywords: 1},]
            file: /home/punck/cvs/Twisted/twisted/python/explorer.py
            name: __call__}
         }}

<ObjectLink of x type instance>:
  {members: {x: 1}
   class: 'twisted.python.explorer.WatchingA8195574'
   methods:
     {foo:
        <ObjectLink of x.foo type instance_method>:
          {class: 'twisted.python.explorer.WatchingA8195574'
           self: '<twisted.python.explorer.WatchingA8195574 instance at
                  0x8195574>'
           doc:
             Pretend to be the method I replaced, and ring the bell.

           line: 651
           signature:
             [{name: instance},
              {name: a
               list: 1},
              {name: kw
               keywords: 1},]
           file: /home/punck/cvs/Twisted/twisted/python/explorer.py
           name: __call__}
         }}
```

TODO: `/watch` might be broken right now.

As you can see, `/watch` really gives you a lot of power (and a lot of output, too – hopefully we'll have a nice GUI display for this in the future). The `/browse` and `/watch` functionality is brought to you by the `twisted.manhole.explorer` module, which was written largely by Kevin Turner.

TODO: Add an example using `twisted.python.rebuild.rebuild`. This lets you tell your application (remotely) to reload its classes, allowing you to upgrade a running server without missing a beat.

Have fun!

2.5 Creating and working with a telnet server

Run `mktap telnet -p 4040 -u admin -w woohoo` 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.protocols.telnet.ShellFactory
Twisted 0.15.5
username: admin
password: ******
>>>
```

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__)
['EverythingEphemeral', 'ServerOptions', '__builtins__', '__doc__', '__name__',
 'application', 'config', 'copyright', 'imp', 'initRun', 'load', 'log',
'logFile', 'logPath', 'logfile', 'main', 'mainMod', 'oldstderr', 'oldstdin',
'oldstdout', 'os', 'platformType', 'rotateLog', 'runtime', 'signal', 'string',
 'styles', 'sys', 'traceback', 'usage', 'util']

>>> __main__.application
<telnet app>
>>> dir(__main__.application)
['authorizer', 'connectors', 'delayeds', 'gid', 'name', 'persistenceVersion',
 'ports', 'resolver', 'running', 'services', 'uid', 'written']
```

From this session we learned that there is an application object stored in `__main__` that's a telnet app, and it has some scary attributes that we're not going to worry about for now.

Alright, so now you've decided that you hate Twisted and want to shut it down. Or you just want to go to bed. Either way, I'll tell you what to do. First, disconnect from your telnet server. Then, back at your system's shell prompt, type `kill `cat twistd.pid`` (the quotes around `cat twistd.pid` are backticks, not single-quotes). If you list the contents of your current directory again, you'll notice that there will be a file named telnet-shutdown.tap. If you wanted to restart the server with exactly the same state as you left it, you could just run `twistd -f telnet-shutdown.tap`. This is why Twisted doesn't need any sort of configuration files – all the configuration data is stored right in the objects!

Now that you've learned how to create a telnet server with 'mktap telnet', we'll delve a little deeper and learn how one is created behind the scenes. Start up a python interpreter and make sure that the 'twisted' directory is in your module search path.

```
Python 1.5.2 (#0, Dec 27 2000, 13:59:38)  [GCC 2.95.2 20000220 (Debian
GNU/Linux)] on linux2
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> import sys
>>> sys.path.append('/twisted/Twisted')
```

I installed Twisted in /twisted, so the place where my 'twisted' package directory is at is /twisted/Twisted/twisted (confusing, I know). For Python to find the 'twisted' package, it must have the directory *containing* the package in sys.path – which is why I added /twisted/Twisted.

```
>>> from twisted.internet import app, tcp
>>> from twisted.protocols import telnet
>>> application = app.Application('telnet')
>>> ts = telnet.ShellFactory()
>>> application.listenTCP(4040, ts)
```

The above is basically what `mktap telnet` does. First we create a new Twisted Application, we create a new telnet Shell Factory, and we tell the application to listen on TCP port 4040 with the ShellFactory we've created.

Now let's start the application. This causes all ports on the application to start listening for incoming connections. This step is basically what the 'twistd' utility does.

```
>>> application.run()
twisted.protocols.telnet.ShellFactory starting on 4040
```

You now have a functioning telnet server! You can connect with your telnet program and work with it just the same as you did before. When you're done using the telnet server, you can switch back to your python console and hit ctrl-C. The following should appear:

```
Starting Shutdown Sequence.
Stopping main loop.
Main loop terminated.
Saving telnet application to telnet-shutdown.tap...
Saved.
>>>
```

Your server was pickled up again and saved to the telnet-shutdown.tap file, just like when you did `kill 'cat twistd.pid'`.

2.5.1 More Complicated Configuration

Let's suppose that we have the following application:

```
#! /usr/bin/python

from twisted.internet.app import Application
from twisted.internet.protocol import Factory
from twisted.protocols.wire import QOTD

app = Application("demo")
```

```
# add QOTD server
f = Factory()
f.protocol = QOTD
app.listenTCP(8123, f)

app.run()
```

Source listing — *manhole1.py*

Once this is running, it would be nice to poke around inside it. We can add the manhole-shell by adding a few lines to create a new server (a Factory) listening on a different point:

```
#! /usr/bin/python

from twisted.internet.app import Application
from twisted.internet.protocol import Factory
from twisted.protocols.wire import QOTD
import twisted.manhole.telnet

app = Application("demo")

# add QOTD server
f = Factory()
f.protocol = QOTD
app.listenTCP(8123, f)

# Add a manhole shell
f = twisted.manhole.telnet.ShellFactory()
f.username = "boss"
f.password = "sekrit"
f.namespace['foo'] = 12
app.listenTCP(8007, f)

app.run()
```

Source listing — *manhole2.py*

With this in place, you can telnet to port 8007, give the username “boss” and password “sekrit”, and you’ll end up with a shell that behaves very much like the Python interpreter that you get by running `python` all by itself, with lines you type prefixed with >>>.

```
% telnet localhost 8007
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
```

```
twisted.manhole.telnet.ShellFactory
Twisted 0.99.2
username: boss
password: *****
>>>
```

Note that the original Quote-Of-The-Day server is still running on port 8123 by using `nc localhost 8123` (or `telnet localhost 8123` if you don't have netcat installed).

```
% nc localhost 8123
An apple a day keeps the doctor away.
```

The initial namespace of the manhole interpreter is defined by a dictionary stored in the '`namespace`' attribute of the ShellFactory. For convenience, you can put references to any objects you like in that dict (`f.namespace['foo'] = 12`), and then retrieve them by name from the telnet session.

```
>>> foo
12
```

Of course we can change that namespace by evaluating expressions in the interpreter. To be a useful debugging tool, however, we want to get access to our servers (the protocol `Factory` instances and everything hanging off of them). We start by gaining access to the main `Application` instance through a global variable stored in the `app` module:

```
>>> import twisted.internet.app
>>> a = twisted.internet.app.theApplication
>>> a
<'demo' app>
```

This object holds three things of interest: the list of Delayeds (functions scheduled to run some number of seconds in the future), the list of Services (subclasses of ApplicationService that have been added to the application, most notably Perspective Broker services), and the list of ports on which protocol Factories are listening. The ports are kept in a list, and the Factory object itself is available inside that list (word wrapped for clarity):

```
>>> a.tcpPorts
[(8123, <twisted.internet.protocol.Factory instance at 0x8249b8c>, 5, ''),
(8007, <twisted.manhole.telnet.ShellFactory instance at 0x824aefc>, 5, '')
]
>>> f = a.tcpPorts[0][1]
>>> f
<twisted.internet.protocol.Factory instance at 0x8249b8c>
```

Now that we have access to that Factory, what can we do? We can modify any attribute of the object, or call functions on it. Remember that the Factory stores a reference to a subclass of Protocol, and it uses that reference to create new Protocol instances for each new connection. We can change that reference to make the Factory create something else:

```
>>> f.protocol
<class twisted.protocols.wire.QOTD at 0x824a66c>
>>> from twisted.protocols.wire import Daytime
>>> f.protocol = Daytime
```

Congratulations, you've just changed the Factory to use the `Daytime` protocol instead of the `QOTD` protocol. You have just transformed the QOTD server into a Daytime server. Connect to port 8123 now and see the difference: you get a timestamp instead of a quote:

```
% nc localhost 8123
Sat Sep 28 09:11:37 2002
```

From here, you can do anything you want to your application. It is a good idea to check the source for the `Application` and `Service` classes to see what else you can extract from them.

Note: to terminate your session, you'll need to exit the telnet or netcat program (the usual control-D that works in the Python interpreter won't work here). Try control-] for telnet. Also note that any exceptions caused by your manhole session will be displayed both in the telnet session *and* in the stderr on the application side.

Chapter 3

High-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 82) 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 72).

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 Using app.Application

3.2.1 Motivation

Calling reactor methods (like `.listenTCP` and `.run`) directly, as in the examples in Writing Servers (page 58), is a good way to immediately demonstrate the use of Factories and Protocols. But you would ask for more from a fully-fledged, easy-to-run, easy-to-configure Internet Server (with capital I and S). To be precise, your users (defined as someone who wants to install your server without knowing all the details of how it works) will ask for more from it. Twisted provides this for you.

What more could we want from our little test program? Well:

- *configuration arguments*:

  suppose your QOTD server behaves a bit more like the normal port 17 server and pulls a random line from `/usr/share/fortunes`. Your QOTDFactory() might take a filename to indicate where the QOTD protocols should pull these lines. It would be nice if the person installing your quote server didn't have to modify any Python code to change where this file should be found.

  Likewise, what if they want it to listen on some other port? That shouldn't require editing the code.

- *starting/stopping and persistence*:

  If your protocol demands that you keep some state from one invocation of the server to the next, you'll need to save some information before the server shuts down, and to restore it again when you start back up.

> Suppose your protocol's purpose in life is to generate one-time keys, and that people can connect to it to retrieve a single-use key. (Don't ask me why they might want to do this. Security is such a weird big thing that chances are somebody out there will want to do something that's probably pretty dumb when you think about it carefully). The important thing is that you never give out the same key twice. So you have to remember a sequence number, and each time you give out a key, you bump up the number. Before you shut down, you save the number to a file somewhere; at start up, if the file exists you read the number from it, if it doesn't exist, you start at 0. (an example is included below)
>
> This kind of persistent data is a common need, and many kinds of servers require it. Hence Twisted provides an easy way to record and reload this data.

This functionality is provided by the Application class (defined in twisted/internet/app.py). You create an Application with a constructor like any other object. Then you tell the app to listen to ports (just like you told the reactor to in the previous example), providing a Factory on each one. The difference is that the App won't starting listening on those ports right away, but will wait until it starts to run.

When you're done setting up the ports, you have two options: you can start running the app immediately, by calling the `.run()` method, or you can save the Application out to a file by calling the `.save()` method. The saved application can then be started later by using the `twistd` utility.

3.2.2 Example Application

Here is a short example of the first option, running the server immediately. This example uses the pre-defined Daytime protocol, which simply sends the current time to each client.:

```
#! /usr/bin/python

import twisted.internet.app
from twisted.protocols.wire import Daytime
from twisted.internet.protocol import Factory

app = twisted.internet.app.Application("daytimer")
f = Factory()
f.protocol = Daytime
app.listenTCP(8813, f)

app.run()
```

Source listing — *app1.py*

This program will start listening to port 8813 in the `app.run()` call, and won't return from that call until the server is terminated (probably when you send it SIGINT).

To use the second option and launch the server later, just use `.save()` instead of `.run()`. The `.save()` method takes a base name for the generated `.tap` file:

```
...
app.listenTCP(8813, f)

app.save("start")
```

When you run this program, it will create a file called `daytime-start.tap`, and then exit. (The name is obtained by combining the application name with the argument to `.save()`). To start the server from the "freeze-dried" `.tap` file, use `twistd` (text wrapped to be more readable):

```
% ./app2.py
Saving daytimer application to daytimer-start.tap...
Saved.
% twistd -f daytimer-start.tap
% tail twistd.log
30/09/2002 01:38 [-] Log opened.
30/09/2002 01:38 [-] twistd 0.99.2 (/usr/bin/python2.2 2.2.1) starting up
30/09/2002 01:38 [-] license user: Nobody <>
30/09/2002 01:38 [-] organization: No Organization
30/09/2002 01:38 [-] reactor class: twisted.internet.default.SelectReactor
30/09/2002 01:38 [-] Loading daytimer-start.tap...
30/09/2002 01:38 [-] Loaded.
30/09/2002 01:38 [*daytimer*] twisted.internet.protocol.Factory starting on 8813
30/09/2002 01:38 [*daytimer*] Starting factory
<twisted.internet.protocol.Factory instance at 0x81ac9fc>
%
```

That will "thaw out" the `.tap` file, create the Application, and then run it just as if you'd invoked app.run() yourself. It forks the new server off into the background (so twistd itself completes instead of waiting for the server to die), writes the server's process ID to a file called `twistd.pid`, and directs all the server's stdout messages to a file called `twistd.log` (these file names can be changed by appropriate arguments to `twistd`: see `twistd -h` for a list).

When you try this example, be aware that `twistd` returns right away, but it takes a second or two for the server to actually start. The `twistd.pid` file won't be created until it does. Wait a moment before doing `ls` or `netstat`, or you'll think that the server failed to start. If it persists in failing, look in `twistd.log` for details. Remember that trying to bind to a reserved port will fail unless you're root, and the exception will be listed at the end of the log file.

To kill the server, just do:

```
% kill `cat twistd.pid`
```

When the server is shut down, you'll notice that it creates a file called `daytimer-shutdown.tap` in the directory it was run from (again, the name is derived from the application name and the word "shutdown"). This `.tap` file is just like the `daytimer-start.tap` created by your original setup program, except that it represents the state of the Application object as it existed just before shutdown, rather than when it was freshly created by your code.

Also note that the `twistd.pid` file is automatically deleted when the application shuts down.

3.2.3 Saving State Across Sessions: Adding Persistent Data

You can add persistent data (like that sequence number described above) to the protocol Factory object, and it will get saved in the `-shutdown.tap` file. Then, if you restart the server with `twistd -f daytimer-shutdown.tap`, the new server will get the data saved by the old server, and it can pick up where the old one left off, as if the server had been running continuously the whole time.

To take advantage of this, simply add the attributes you want to the Factory, or to your subclass of Service (see the docs on Perspective Broker for details about Services). When the application terminates, it simply pickles up the whole Application (and everything it references, including Factories and Services). Any attributes or objects you have added will be saved and later restored.

Here is an example:

```
#! /usr/bin/python

from twisted.internet.protocol import Protocol, Factory

class OneTimeKey(Protocol):
    def connectionMade(self):
        key = self.factory.nextkey
        print "giving key", key
        self.factory.nextkey += 1
        self.transport.write("%d\n" % key)
        self.transport.loseConnection()

def main():
    # namespaces are weird. See the comment in doc/examples/echoserv.py
    import app3
    from twisted.internet.app import Application
    f = Factory()
    f.protocol = app3.OneTimeKey
    f.nextkey = 0
    app = Application("otk")
    app.listenTCP(8123, f)
    app.save("start")


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

Source listing — *app3.py*

To demonstrate this, do the following:

```
% ./app3.py
Saving otk application to otk-start.tap...
Saved.
% twistd -f otk-start.tap
%
% nc localhost 8123
0
% nc localhost 8123
1
% nc localhost 8123
```

```
2
%
```

Note that the stdout of the process is being directed into the log file, contained in `twistd.log`. Now stop the server, verify that it is no longer running, then restart it from the saved-at-shutdown `.tap` file:

```
% kill `cat twistd.pid `
% nc localhost 8123
localhost [127.0.0.1] 8123 (?) : Connection refused
% twistd -f otk-shutdown.tap
% nc localhost 8123
3
%
```

Notice how the saved `.nextkey` attribute was restored, and the application picks up where it left off.

3.2.4 Configuration arguments

To do this right, you'll want to follow the sequence described by the writing plugins (page 40) document. Instead of writing a short program that creates a `.tap` file (by creating an Application, doing various .listenTCPs on it, then calling .save), you will write a subroutine called updateApplication(). This subroutine should take a bunch of config arguments (using the usage.Options class described in the plugins document) and use them to create Factories and feed them to .listenTCP on an *existing* Application instance.

With that in place, and a few files to register this new server you've created, a utility program called '`mktap`' can relieve you of the business of gathering user arguments and creating the app instance. `mktap` can use the Options subclass you define in your build-a-tap class to figure out what arguments are legal (`--port` taking a number, `--quotes` taking a filename, etc), provide `--help` with a list of valid arguments, and parse everything the user passes in `argv[]`. It creates the Application, then passes the app and the parsed options to your `updateApplication()` method, where you do the server-specific creation of a Factory and the various `listenTCP` calls. Then `mktap` saves out the `.tap` file, ready for starting by `twistd`.

The end result is that installing your new server is simplified to the following steps:

- Unpack your server module (including the classes and plugin glue files) into somewhere on your PYTHONPATH, perhaps `/usr/local/lib/python`.
- Run the standard `mktap` program, giving it the name of your module and whatever configuration arguments it requires. Watch it create a `.tap` file.
- Use `twistd` to start the server contained in the `.tap` file.

Pretty easy. At least your users will think so.

And, once your application is defined by the `.tap` file, there are other tools that can be used to configure it. `tap2deb` is a tool that creates installable Debian `.deb` packages from your `.tap` file, making installation even easier.

The Application object has some other features designed to solve common server needs:

- logging is controlled, through the log.Logger class
- delayed events can be scheduled, with the .addDelayed() method

- the process can switch to a different uid/gid after binding reserved ports
- styles.Versioned allows old saved copies of an object to be upgraded when new versions of the class are available
- Applications have Authorizers, used to authenticate client connections
- Applications have Services, which can be accessed by PB clients

3.3 Writing a New Plug-In for Twisted

3.3.1 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 the "right way" to get started working on a Twisted application. It probably won't answer your specific questions about how to do things like schedule functions to call in the future (page 81) or listen on a socket (page 58); there are other documents that address these concerns and you can read them later. *Although there are other ways for Twisted to call your code, all Twisted projects should start as a plug-in of some kind.*

3.3.2 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 class that you will deal with, both as a Twisted developer and administrator, is `twisted.internet.app.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 than defining methods on 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 event handlers 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.

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.

3.3.3 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"[3] can define mechanisms for extensibility using plug-ins. Two Twisted dot-products already load such plug-ins. The `twisted.tap`package loads Twisted Application builder modules (TAP plug-ins) and the `twisted.coil` package loads configuration modules (COIL plug-ins).

Twisted finds its plug-ins by using pre-existing Python concepts; the load path, and packages. Every top-level Python package[4] (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])
```

[3] http://twistedmatrix.com/products/dot-products

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

- `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).
- `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.

3.3.4 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[5] 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.
```

[5] 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 72). (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
from twisted.internet.app import Application

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 58) 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 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 updateApplication(app, config):
    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.
    app.listenTCP(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 `updateApplication(app, config)`, which will be passed an instance of a `twisted.internet.app.Application` and 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 `updateApplication` to make the actual connections between objects.

A more detailed discussion of `twisted.python.usage.Options` can be found in the document Using `usage.Options` (page 51).

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[6] 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]

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
```

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

```
                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:
    coil            A web-based configuration manager.
    ftp             An FTP server.
    im              A multi-protocol chat client.
    inetd
    issues          Bug reporting/tracking service.
    mail            An email service.
    manhole         An interactive remote debugger service.
    news            News Server
    parent          Parent service.
    pinger          Zoot Pinger TAP builder module
    ponger          Zoot Ponger TAP builder module
    portforward     A simple port-forwarder.
    qotd            Example of a TAP builder module.
    sister          Sister service.
    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.
    zoot            Zoot TAP builder module
```

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!

3.4 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 web-site 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.

3.4.1 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.

3.4.2 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] )
```

3.4.3 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.

3.4.4 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.

3.4.5 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.

3.4.6 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)
```

3.4.7 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)
```

3.5 Using usage.Options

3.5.1 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
```

3.5.2 Boolean Options

For simple boolean options, define the attribute `optFlags` like this:

```
class Options(usage.Options):

    optFlags = [["fast", "f"], ["safe", "s"]]
```

`optFlags` should be a list of 2-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 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"], ["good", "g"], ["cheap", "c"]]

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"]]

class SpecificOptions(BaseOptions):

    optFlags = [["fast", "f"], ["good", "g"], ["cheap", "c"]]
```

Is the same as:

```
class SpecificOptions(BaseOptions):

    optFlags = [["quiet", "q"], ["fast", "f"], ["good", "g"], ["cheap", "c"]]
```

3.5.3 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"], ["good", "g"], ["cheap", "c"]]
    optParameters = [["user", "u", None]]

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.

3.5.4 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], ['vendor', 'v', None],
                     ['release', 'r', None]]

class CheckoutOptions(usage.Options):
    optParameters = [['module', 'm', None], ['tag', 'r', None]]

class Options(usage.Options):
    subCommands = [['import', None, ImportOptions],
                   ['checkout', None, CheckoutOptions]]

    optParameters = [['compression', 'z', 0], ['repository', 'r', None]]
```

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.

3.5.5 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.

3.5.6 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]]

    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)`.

3.5.7 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"], ["good", "g"], ["cheap", "c"]]

    def postOptions(self):
        if self['fast'] and self['good'] and self['cheap']:
            raise usage.UsageError, "can't have it all, brother"
```

3.6 DirDBM: Directory-based Storage

3.6.1 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()
[]
```

3.6.2 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]
```

Chapter 4

Low-Level Twisted

4.1 Reactor Basics

The reactor is the core of the event loop within Twisted and provides a basic interface to a number of services, including network communications, threading, and event dispatching.

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 84).

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 66) document for more information.
- `IReactorSSL`: SSL networking support.
- `IReactorTCP`: TCP networking support. More information can be found in the Writing Servers (page 58) and Writing Clients (page 62) documents.
- `IReactorThreads`: Threading use and management. More information can be found within Threading In Twisted (page 82).
- `IReactorTime`: Scheduling interface. More information can be found within Scheduling Tasks (page 81).
- `IReactorUDP`: UDP networking support. More information can be found within UDP Networking (page 65).
- `IReactorUNIX`: UNIX socket support.

4.2 Writing Servers

4.2.1 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 40). This document is only relevant to TCP, SSL and Unix socket servers, there is a separate document (page 65) 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.

4.2.2 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 40) 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.

4.2.3 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(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 35).

4.3 Writing Clients

4.3.1 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 65).

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.

4.3.2 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.

4.3.3 ClientFactory

With the new API, Protocols no longer connect directly using reactor.client*. Instead, 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 startedConnection(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()
```

4.3.4 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. 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.

Reconnection

Many times, the connection of a client will be lost unintentionally due to network errors. In the case of the ircLogBot, leaving the bot disconnected will result in the loss of the log data until the administrator reconnects the bot. However, with the new API this can be automated. The relevant part of ircLogBot.py follows:

```
from twisted.internet import protocol
class LogBotFactory(protocol.ClientFactory):

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

That last line is the most important. 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.

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 ircLogBot.py: (`LogBot.log()` just logs the data to the file object stored in `LogBot.file`)

```
from twisted.internet import protocol
from twisted.protocols import irc
class LogBot(irc.IRCClient):

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

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

class LogBotFactory(protocol.ClientFactory):

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

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.

4.4 UDP Networking

4.4.1 Overview

Unlike TCP, UDP has no notion of connections. A UDP socket can receive datagrams from any server on the network, and send datagrams to any host on the network. In addition, datagrams may arrive in any order, never arrive at all, or be duplicated in transit.

Since there are no 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.

4.4.2 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.

4.4.3 Connected UDP

A connected UDP socket is slighly different from a standard one - it can only send and receive datagramss 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.

The protocol's `transport` attribute will implement the `twisted.internet.interfaces.IUDPConnectedTransport` interface.

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

class Echo(ConnectedDatagramProtocol):

    def datagramReceived(self, data):
        self.transport.write(data)

reactor.connectUDP("www.example.com", 9999, Echo())
reactor.run()
```

4.5 Using Processes

4.5.1 Overview

Along with connection to servers across the internet, Twisted also connects to local processes with much the same API. The API is described in more detail in the documentation of:

- `twisted.internet.interfaces.IReactorProcess`
- `twisted.internet.interfaces.IProcessTransport`
- `twisted.internet.protocol.ProcessProtocol`

4.5.2 Running Another Process

Processes are run through the reactor, using `reactor.spawnProcess()`. Pipes are created to the child process, and added to the reactor core so that the application will not block while sending data into or pulling data out of the new process. `reactor.spawnProcess()` requires two arguments, processProtocol and executable, and optionally takes 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)
```

- `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.

`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).

`reactor.spawnProcess()` returns an instance that implements the `twisted.internet.interfaces.IProcessTransport`.

4.5.3 Writing a ProcessProtocol

The ProcessProtocol you pass to spawnProcess is your interaction with the process. It has a very similar signature to a regular Protocol, but it has several extra methods to deal with events specific to a process. In our example, we will interface with 'wc' to create a word count of user-given text. First, we'll start by importing the required modules, and writing the initialization for our ProcessProtocol.

```
from twisted.internet import protocol
class WCProcessProtocol(protocol.ProcessProtocol):
```

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

When the ProcessProtocol is connected to the protocol, it has the connectionMade method called. In our protocol, we will write our text to the standard input of our process and then close standard input, to 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()
```

4.5.4 Things that can happen to your ProcessProtocol

These are the methods that you can usefully override in your subclass of `ProcessProtocol`:

- `.connectionMade`: This is called when the program is started, and makes a good place to write data into the stdin pipe (using `self.transport.write()`).

- `.outReceived(data)`: This is called with data that was received from the process' stdout pipe. Pipes tend to provide data in larger chunks than sockets (one kilobyte is a common buffer size), so you may not experience the "random dribs and drabs" behavior typical of network sockets, but regardless you should be prepared to deal if you don't get all your data in a single call. To do it properly, `outReceived` ought to simply accumulate the data and put off doing anything with it until the process has finished.

- `.errReceived(data)`: This is called with data from the process' stderr pipe. It behaves just like `outReceived`.

- `.inConnectionLost`: This is called when the reactor notices that the process' stdin pipe has closed. Programs don't typically close their own stdin, so this will probably get called when your ProcessProtocol has shut down the write side with `self.transport.loseConnection()`.

- `.outConnectionLost`: This is called when the program closes its stdout pipe. This usually happens when the program terminates.

- `.errConnectionLost`: Same as `outConnectionLost`, but for stderr instead of stdout.

- `.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. XXX: check twisted/internet/process.py:v1.30:line357, I think death-by-signal wouldn't be reported properly.

  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.

4.5.5 Things you can do from your ProcessProtocol

The following are the basic ways to control the child process:

- `self.transport.write(data)`: Stuff some data in the stdin pipe. Note that this `write` method will queue any data that can't be written immediately. Writing will resume in the future when the pipe becomes writable again.

- `self.transport.closeStdin`: Close the stdin pipe. Programs which act as filters (reading from stdin, modifying the data, writing to stdout) usually take this as a sign that they should finish their job and terminate. For these programs, it is important to close stdin when you're done with it, otherwise the child process will never quit.

- `self.transport.closeStdout`: Not usually called, since you're putting the process into a state where any attempt to write to stdout will cause a SIGPIPE error. This isn't a nice thing to do to the poor process.

- `self.transport.closeStderr`: Not usually called, same reason as `closeStdout`.

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

4.5.6 Verbose Example

Here is an example that is rather verbose about exactly when all the methods are called. It writes a number of lines into the `wc` program and then parses the output.

```
#! /usr/bin/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 on 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.
%
```

4.5.7 Doing it the Easy Way

Frequently, one just need a simple way to get all the output from a program. For those 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 need to get just the final exit code, the `twisted.internet.utils.getProcessValue` function is useful. Here is an example:

```
from twisted.internet import utils, reactor

def printTrueValue(val):
    print val
    output = utils.getProcessValue('false')
    output.addCallback(printFalseValue)

def printFalseValue(val):
    print val
    reactor.stop()

output = utils.getProcessValue('true')
output.addCallback(printTrueValue)
reactor.run()
```

Source listing — *trueandfalse.py*

4.6 Deferring Execution

4.6.1 The Problem

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):
        self.blockUntilResult(x)
        return result

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

Don't Call Us, We'll Call You

Twisted can not 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. Looking at how this is usually implemented will help us understand the necessity for Deferreds.

```
class Getter:

    def getData(self, x, callback):
        self.callback = callback
        # this call does not block, it ensure self.gotResult is called
        # when we have the result
        self.onResult(x, self.gotResult)

    def gotResult(self, result):
        self.callback(result)

def gotData(d):
    print d

g = Getter()
g.getData(3, gotData)
```

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. There is no way to distinguish between different calls to gotData from different sessions. `Deferred` solves these problems, by creating a single, unified way to defer execution of code that depends on blocking calls.

4.6.2 Deferreds

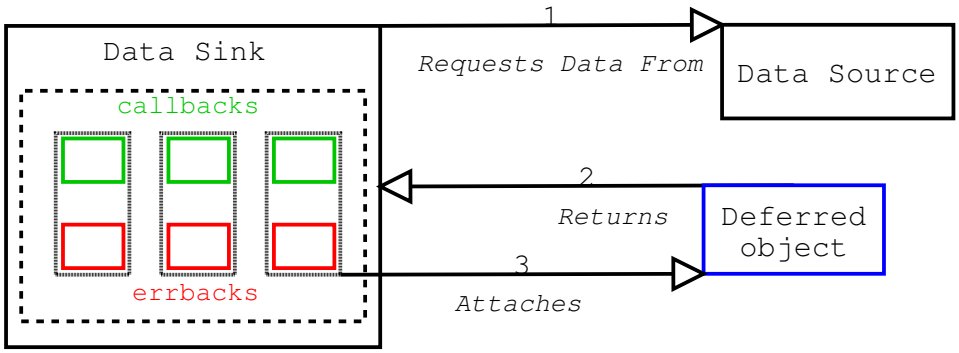
A `twisted.internet.defer.Deferred` is a promise that a function will at some point have a result. We can attach callback functions to a Deferred, and once it gets a result these callbacks will be called. In addition Deferreds allow the developer to register a callback for an error, with the default behavior of logging the error. This is an asynchronous equivalent of the common idiom of blocking until a result is returned or an exception it raised.

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.

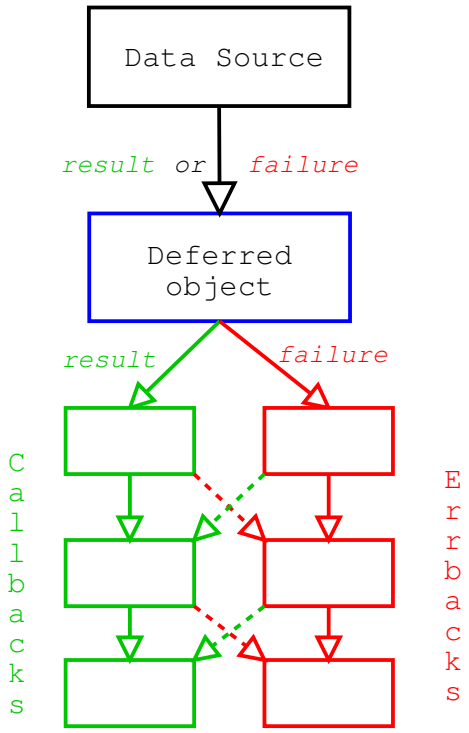
```
import sys
from twisted.internet import defer

class Getter:

    def getResult(self, x):
```

```
        self.d = defer.Deferred()
        self.doNonblockingStuff(x)
        return self.d

    def gotResult(self, result):
        """Called when we get some info from somewhere via the event loop.

        E.g. this may be called because we got a chunk of data off a socket.
        """
        if self.goodResult(result):
            # tell the Deferred that we have a result for it
            self.d.callback(result)
        else:
            # tell the Deferred that we have an error
            self.d.errback("An error has occured.")

def printData(d): sys.stdout.write(d)
def printError(e): sys.stderr.write(e)

g = Getter()
d = g.getResult(3) # notice how this is similar to the blocking version
d.addCallback(printData) # printData will be called when a result is available
d.addErrback(printError) # printError will be called on an error

# run main event loop here
from twisted.internet import reactor
reactor.run()
```

Visual Explanation



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



1. When the result is ready, give it to the Deferred object. `.callback(result)` if the operation succeeded, `.errback(failure)` if it failed. Note that `failure` is typically an instance of a `twisted.python.failure.Failure` instance.

2. Deferred object triggers previously-added (call/err)back with the `result` or `failure`. Execution then follows the following rules, going down the chain of callbacks to be processed.

   - Result of the callback is always passed as the first argument to the next callback, creating a chain of processors.
   - If a callback raises an exception, switch to errback.
   - An unhandled failure gets passed down the line of errbacks, this creating an asynchronous analog to a series to a series of `except:` statements.
   - If an errback doesn't raise an exception or return a `twisted.python.failure.Failure` instance, switch to callback.

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-processed 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 convenience 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)
```

```
d.addErrback(errback1)
d.addCallback(callback2)
d.addErrback(errback2)

# Case 2
d = getDeferredFromSomewhere()
d.addCallbacks(callback1, errback1)
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.

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)
```

4.6.3 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, errback Keywords, asDefaults])`

  This is the method with which 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, **method Keywords)`. If your method is passed in the callback slot, for example, all arguments in the tuple `callback Args` will be passed as `*methodArgs` to your method.

  There exist 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.
  - `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

While deferreds greatly simplify the process of writing asynchronous code by providing a standard for registering callbacks, 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 the curious: the code for `twisted.web.widgets` has a textbook example of this.) 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.

4.6.4 DeferredList

Sometimes you want to be notified after several different events have all happened, rather than individually waiting for each one. 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 also add the Deferreds later:

```
dl.addDeferred(deferred4)
```

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. TODO: add picture here: three columns of callback chains, with a value being snarfed out of the middle of each and handed off to the DeferredList

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` and `fireOnOneErrback`. 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.

4.7 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 in 3.5 seconds: %s" % s

reactor.callLater(3.5, f, "hello, world")
```

If we want a task to run every X seconds repeatedly, we can just re-add it every time it's run:

```
from twisted.internet import reactor

def runEverySecond():
    print "a second has passed"
    reactor.callLater(1, runEverySecond)

reactor.callLater(1, runEverySecond)
```

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()
```

4.8 Using Threads in Twisted

4.8.1 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.

4.8.2 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
```

4.8.3 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")
```

4.8.4 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)
```

4.8.5 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(20)
```

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

4.9 Choosing a Reactor and GUI Toolkit Integration

4.9.1 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 84)
- The poll()-based reactor (page 85)

Platform-specific reactor implementations exist for:

- cReactor for Unix (page 85)
- KQueue for FreeBSD (page 85)
- Java (page 85)
- Win32 (page 86)

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+ (page 86)
- Qt (page 86)
- Tkinter (page 86)
- WxPython (page 87)
- Win32 (page 86)

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()`.

4.9.2 Reactor Functionality

4.9.3 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()
```

	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
Java	Y	N	N	Y	N	Y	Java 1.1+
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 4.1: Summary of reactor features

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()
```

4.9.4 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. Be sure to see the installation notes (page 19) prior to using the cReactor.

```
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()
```

Java

The Java Reactor allows Twisted to run under Jython[1]. It does not currently support AWT or Swing integration.

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

[1] http://www.jython.org/

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()
```

4.9.5 GUI Integration Reactors

GTK+

Twisted integrates with PyGTK[2]. Sample applications using GTK+ and Twisted are available in the Twisted CVS.

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

4.9.6 Non-Reactor GUI Integration

Tkinter

The support for Tkinter[3] 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)
```

An example Twisted application that uses Tk can be found in `twisted/words/ui/tkim.py`.

[2] http://www.daa.com.au/~james/pygtk/
[3] http://www.python.org/topics/tkinter/

wxPython

As with Tkinter (page 86), the support for integrating Twisted with a wxPython[4] application uses specialized support code rather than a simple reactor.

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

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

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

[4] http://www.wxpython.org

Chapter 5

Perspective Broker

5.1 Introduction to Perspective Broker

5.1.1 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.

5.1.2 Class Roadmap

To start with, here are the major classes 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.

- *`Application`*: `internet/app.py`
- *`Service`*: `spread/pb.py`, subclassed from `Service` in `cred/service.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.

- *MultiService*: internet/app.py
- *Factory*: internet/protocol.py
- *BrokerFactory*: spread/pb.py
- *Broker*: spread/pb.py
- *AuthRoot* : 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 that get involved when you start to care about authorization and security:

- *Authorizer*: cred/authorizer.py
- *Identity*: cred/identity.py
- *Perspective*: spread/pb.py, subclassed from Perspective in cred/perspective.py

Subclassing

Technically you can subclass anything you want, but techically you could also write a whole new framework, which would just waste a lot of time. Knowing which classes are useful to change (by making subclasses) is one of the bits of knowledge you pick up after using Twisted for a few weeks. Here are some hints to get started:

- Protocol: subclass this if you need to implement a new protocol on the wire, like HTTP or SMTP (except that almost all of the standard ones are already implemented). You might also subclass one of the standard implementations if you want to change its back-end behavior: make an SMTP server which actually stores the messages in files instead of mailing them, or a Finger server that returns random messages instead of current login status.
- pb.Root, pb.Referenceable: you'll subclass these to make remotely-referenceable objects 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.Perspective, pb.Service: you'll probably end up subclassing these when you get into PB programming (with authorization). There are a few methods you'll change, especially with regards to creating new Perspectives.
- Authorizer: subclass this if you want to get users from /etc/passwd, or a database, or LDAP, or other list of usernames and passwords.

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

5.1.3 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.Perspective`

  This is the first class we dealt with. Perspectives are slightly special because they are the root object that a given user can access from a service. 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 a perspective directly.

  Remotely accessible methods on Perspectives 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.

5.1.4 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 user who is requesting it. The `perspective` argument will be an instance of the `Perspective` subclass for your service, the one which is either pasing 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` passes a `Perspective` instance from your service. It also passes 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.

5.2 Using Perspective Broker

5.2.1 Basic Example

The first example to look at is a complete (although somewhat trivial) application. It uses `BrokerFactory()` on the server side, and `pb.getObjectAt()` on the client side.

```
from twisted.spread import pb
from twisted.internet import app
class Echoer(pb.Root):
    def remote_echo(self, st):
        print 'echoing:', st
        return st
if __name__ == '__main__':
    appl = app.Application("pbsimple")
    appl.listenTCP(8789, pb.BrokerFactory(Echoer()))
    appl.run()
```

Source listing — *pbsimple.py*

```
from twisted.spread import pb
from twisted.internet import reactor
def gotObject(object):
    print "got object:",object
    object.callRemote("echo", "hello network").addCallback(gotEcho)
def gotEcho(echo):
    print 'server echoed:',echo
    reactor.stop()
def gotNoObject(reason):
    print "no object:",reason
    reactor.stop()
pb.getObjectAt("localhost", 8789, 30).addCallbacks(gotObject, gotNoObject)
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.BrokerFactory()`. 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.BrokerFactory()` 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 `BrokerFactory` is given the root object, and the client retrieves it.

The client side calls `pb.getObjectAt` to make a connection to a given port. This is a convenience function (not a method) which runs through the PB protocol steps necessary to retrieve the root object from a `BrokerFactory` sitting at the given port.

Because `.getObjectAt()` has to make a network connection 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 72) for a complete explanation of `Deferred`s). If and when the connection succeeds and a reference to the remote root object is obtained, this callback is run. The first argument passed to the callback is a remote reference to the distant root object. (you can give other arguments to the callback too, see the other parameters for `.addCallback()` and `.addCallbacks()`).

The callback does:

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

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

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

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

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

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

The remote reference object (the one given to `getObjectAt()`'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` 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.

[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.

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 `BrokerFactory`.)

5.2.2 Complete Example

A service is the "global" state associated with your application, which can contain things such as support for archiving objects, basic abstractions common to all users, and collections of domain-specific objects. A perspective is the representation of a user with respect to a particular service. For PB, a Perspective is where all interaction begins. When a user logs in for the first time, all the methods they can initially call are methods of their Perspective. The Perspective's methods can return objects which themselves have methods that you can call, as well as copies of objects, as described later.

```
from twisted.spread import pb

class QuoteReader(pb.Perspective):
    def perspective_nextQuote(self):
        return self.service.quoter.getQuote()

class QuoteService(pb.Service):
    def __init__(self, quoter, serviceName, serviceParent, authorizer):
        pb.Service.__init__(self, serviceName, serviceParent, authorizer)
        self.quoter = quoter
    perspectiveClass = QuoteReader
```

Quote Service and Perspective — *pbquote.py*

For examples of these, we're returning to the TwistedQuotes project discussed in the "Writing Plugins". The PB `Service` for TwistedQuotes is pretty small. The only thing it needs to keep track of for itself is the quoter object; PB's service, that we will inherit from, already keeps track of perspectives.

The perspective is a `QuoteReader`, which publishes one method. By subclassing `Perspective`, we are declaring that all methods with the `perspective_` prefix are remotely accessible.

In order to get this Service 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 PB broker factory 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.python import usage        # twisted command-line processing
from twisted.spread import pb           # Perspective Broker
from twisted.cred import authorizer     # cred authorizer, to allow logins
```

```
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 updateApplication(app, config):
    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:
        auth = authorizer.DefaultAuthorizer(app)
        pbserv = pbquote.QuoteService(quoter, "twisted.quotes", app, auth)
        # create a quotereader "guest" give that perspective a password and
        # create an account based on it, with the password "guest".
        pbserv.createPerspective("guest").makeIdentity("guest")
        pbfact = pb.BrokerFactory(pb.AuthRoot(auth))
        app.listenTCP(int(pbport), pbfact)
    app.listenTCP(port, factory)
```

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

In the TAP builder, we create a `QuoteService` that wraps the quoter. We then create a `QuoteReader` perspective and attach it to the `QuoteService`, through the `createPerspective` call inherited from `Service`. Finally, we register with the `QuoteService`'s authorizer.

Accessing this through a client is fairly easy, as we can use the `pb.connect` convenience function.

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

def connected(perspective):
    perspective.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()

pb.connect("localhost", # host name
           pb.portno, # port number
           "guest", # identity name
           "guest", # password
           "twisted.quotes", # service name
           "guest", # perspective name (usually same as identity)
           None, # client reference, used to initiate server->client calls
           30 # timeout of 30 seconds before connection gives up
           ).addCallbacks(connected, # what to do when we get connected
                          failure) # and what to do when we can't

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

PB Quotes Client Code — *pbquoteclient.py*

`pb.connect` will handle all the details of creating a connection and authenticating. It returns a `Deferred`, which will have its callback called when `pb.connect` connects to a perspective, and have its `errback` called when the object-connection fails for any reason, whether it's host lookup failure, connection refusal, or incorrect authentication credentials.

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
Saving qotd application to qotd.tap...
Saved.
% twistd -f qotd.tap
% python -c 'import TwistedQuotes.pbquoteclient'
An apple a day keeps the doctor away.
```

The argument to this callback, `perspective`, is a `RemoteReference`. The `perspective` reference represents a reference to a `QuoteReader` perspective 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.connect`, 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 othe building blocks you can use.

5.2.3 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
import twisted.internet.app

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

app = twisted.internet.app.Application("pb1server")
app.listenTCP(8800, pb.BrokerFactory(One()))
app.run(save=0)
```

Source listing — *pb1server.py*

```
#! /usr/bin/python

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

def main():
    def1 = pb.getObjectAt("localhost", 8800, 30)
    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].

5.2.4 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
import twisted.internet.app

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
```

[3] This can be overridden, by subclassing one of the Serializable flavors and defining custom serialization code for your class. See XXX 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.

```
        if two == newtwo:
            print "One.checkTwo(): our twos are the same"


app = twisted.internet.app.Application("pb2server")
two = Two()
root_obj = One(two)
app.listenTCP(8800, pb.BrokerFactory(root_obj))
app.run(save=0)
```

Source listing — *pb2server.py*

```
#! /usr/bin/python

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

def main():
    foo = Foo()
    pb.getObjectAt("localhost", 8800, 30).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 72) 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.

5.2.5 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
import twisted.internet.app

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)

app = twisted.internet.app.Application("pb3server")
app.listenTCP(8800, pb.BrokerFactory(One()))
app.run(save=0)
```

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()
    def1 = pb.getObjectAt("localhost", 8800, 30)
    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.

5.2.6 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`. (You do define `Exception` subclasses, right? String exceptions are, like, *so* 5 minutes ago. Get with the new century, ok?). 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

from twisted.spread import pb
import twisted.internet.app

class MyError(pb.Error):
```

[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.

```
    """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():
    app = twisted.internet.app.Application("exc_server")
    app.listenTCP(8800, pb.BrokerFactory(One()))
    app.run(save=0)

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

Source listing — *exc_server.py*

```
#! /usr/bin/python

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

def main():
    d = pb.getObjectAt("localhost", 8800, 30)
    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`.

5.2.7 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.app import Application
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()

app = Application("trap_server")
app.listenTCP(8800, pb.BrokerFactory(One()))
app.run(save=0)
```

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)

deferred = pb.getObjectAt("localhost", 8800, 30)
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 eat breaking space [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 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`.

[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.

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 XXX for an example.

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.DeadReference Error`. 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.

5.3 PB Copyable: Passing Complex Types

5.3.1 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`).

5.3.2 Motivation

From the previous chapter (page 92), 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.

5.3.3 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

[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

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.

5.3.4 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

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):
```

into a local object of type B. It is these objects "B" that are the "Unjellyable" second argument of the `setUnjellyableForClass` function.

In particular, "unjellyable" does *not* mean "cannot be jellied". `Unpersistable` means "not persistable", but "unjelly", "unserialize", and "unpickle" mean to reverse the operations of "jellying", "serializing", and "pickling".

```
        print "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)
    deferred = pb.getObjectAt("localhost", 8800, 30)
    deferred.addCallback(sender.got_obj)
    reactor.run()

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

Source listing — *copy_sender.py*

```
#! /usr/bin/python

from twisted.internet.app import Application
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()

app = Application("copy_receiver")
app.listenTCP(8800, pb.BrokerFactory(Receiver()))
app.run(save=0)
```

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:

```
% ./copy_receiver.py
twisted.spread.pb.BrokerFactory starting on 8800
Starting factory <twisted.spread.pb.BrokerFactory instance at 0x815085c>
   [program pauses here until copy_sender.py is run]
 got pond: <__main__.ReceiverPond instance at 0x832941c>
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. 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)
    deferred = pb.getObjectAt("localhost", 8800, 30)
    deferred.addCallback(sender.got_obj)
    reactor.run()

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

Source listing — *copy2_sender.py*

```
#! /usr/bin/python

from twisted.internet.app import Application
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()

app = Application("copy_receiver")
app.listenTCP(8800, pb.BrokerFactory(Receiver()))
app.run(save=0)
```

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:

```
% ./copy2_receiver.py
twisted.spread.pb.BrokerFactory starting on 8800
Starting factory <twisted.spread.pb.BrokerFactory instance at 0x8337f2c>
 got pond: <copy2_classes.ReceiverPond instance at 0x8150dbc>
 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.

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.

5.3.5 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

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

[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.

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* 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)
```

[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.

```
    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)
    deferred = pb.getObjectAt("localhost", 8800, 30)
    deferred.addCallback(sender.phase1)
    reactor.run()

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

Source listing — *cache_sender.py*

```
#! /usr/bin/python

from twisted.internet.app import Application
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()

app = Application("copy_receiver")
app.listenTCP(8800, pb.BrokerFactory(Receiver()))
app.run(save=0)
```

Source listing — *cache_receiver.py*

When run, this example emits the following:

```
% ./cache_receiver.py
 cache - sitting, er, setting ducks
got pond: <cache_classes.SlaveDuckPond instance at 0x82a15e4>
[2] ducks:  ['one duck', 'two duck']
 cache - addDuck
[3] ducks:  ['one duck', 'two duck', 'ugly duckling']
 cache - removeDuck
[2] ducks:  ['two duck', 'ugly duckling']
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 `callRemote`s to throw away such an error. This is a useful idiom:

[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.

```
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.
- A `perspective` argument is available to `getStateToCacheAndObserveFor`, as well as `stopped Observing`. 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.

5.4 Authentication with Perspective Broker

5.4.1 Motivation

In the examples shown in Using Perspective Broker (page 92) there were some problems. You had to trust the user when they said their name was "bob": no passwords or anything. If you wanted a direct-send one-to-one message feature, you might have implemented it by handing a User reference directly off to another User. (so they could invoke `.remote_sendMessage()` on the receiving User): but that lets them do anything else to that user too, things that should probably be restricted to the "owner" user, like `.remote_joinGroup()` or `.remote_quit()`.

And there were probably places where the easiest implementation was to have the client send a message that included their own name as an argument. Sending a message to the group could just be:

```
class Group(pb.Referenceable):

# ...

  def remote_sendMessage(self, from_user, message):
    for user in self.users:
      user.callRemote("sendMessage", "[%s]: %s" % (from_user, message))
```

But obviously this lets users spoof each other: there's no reason that Alice couldn't do:

```
remotegroup.callRemote("sendMessage", "bob", "i like pork")
```

much to the horror of Bob's vegetarian friends.

(In general, learn to get suspicious if you see `groupName` or `userName` in the argument list of a remotely-invokable method).

You could fix this by adding more classes (with fewer remotely-invokable methods), and making sure that the reference you give to Alice won't let her pretend to be anybody else. You'd probably give Alice her own object, with her name buried inside:

```
class User(pb.Referenceable):
  def __init__(self, name):
    self.name = name
  def remote_sendMessage(self, group, message):
    g = findgroup(group)
    for user in g.users:
      user.callRemote("sendMessage", "[%s]: %s" % (self.name, message))
```

This improves matters because, as long as Alice only has a reference to *this* object and nobody else's, she can't cause a different `self.name` to get used. Of course, you have to make sure that you don't give her a reference to the wrong object.

> **Note:**
> Third party references (there aren't any)
>
> Note that the reference that the server gives to a client is only useable by that one client: if they try to hand it off to a third party, they'll get an exception (XXX: which? looks like an assert in pb.py:290 RemoteReference.jellyFor). This helps 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.
>
> Also note that 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 the 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 id guessing by a malicious client will give them anything else. The dict goes away when the connection is dropped, limiting further the scope of those references.
>
> Of course, 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. A better design is 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.

But now she could sneak into another group. So you might have to have an object per-group-per-user:

```
class UserGroup(pb.Referenceable):
  def __init__(self, group, user):
    self.group = group
    self.user = user
```

```
  def remote_sendMessage(self, message):
    name = self.user.name
    for user in self.group.users:
      user.callRemote("sendMessage", "[%s]: %s" % (name, message))
```

But that means more code, and more code is bad, especially when it's a common problem (everybody designs with security in mind, right? Right??).

So we have a security problem. We need a way to ask for and verify a password, so we know that Bob is really Bob and not Alice wearing her "Hi, my name is Bob" t-shirt. And it would make the code cleaner (i.e.: fewer classes) if some methods could know reliably *who* is calling them.

5.4.2 A sample application

As a framework for this chapter, we'll be referring to a hypothetical game implemented by several programs using the Twisted framework. This game will have multiple players, where users log in using their client programs, and there is a server, and users can do some things but not others[12].

The players make moves in this game by invoking remote methods on objects that live in the server. The clients can't really be relied upon to tell the server who they are with each move they make: they might get it wrong, or (horrors!) lie to mess up the other player.

Let's simplify it to a server-based game of Go (if that can be considered simple). Go has two players, white and black, who take turns placing stones of their own color at the intersections of a 19x19 grid. If we represent the game and board as an object in the server called Game, then the players might interact with it using something like this:

```
class Game(pb.Referenceable):
  def remote_getBoard(self):
    return self.board # a dict, with the state of the board
  def remote_move(self, playerName, x, y):
    self.board[x,y] = playerName
```

"But Wait", you say, yes that method takes a playerName, which means they could cheat and move for the other player. So instead, do this:

```
class Game(pb.Referenceable):
  def remote_getBoard(self):
    return self.board # a dict, with the state of the board
  def move(self, playerName, x, y):
    self.board[x,y] = playerName
```

and move the responsibility (and capability) for calling Game.move() out to a different class. That class is a `pb.Perspective`.

5.4.3 Perspectives

`pb.Perspective` (and some related classes: Identity, Authorizer, and Service) is a layer on top of the basic PB system that handles username/password checking. The basic idea is that there is a separate Perspective object (probably

[12] There actually exists such a thing. It's called twisted.reality, and was the whole reason Twisted was created. I haven't played it yet: I'm too afraid.

a subclass you've created) for each user[13], and *only* the authorized user gets a remote reference to that Perspective 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.

Your code can then look like this:

```
class Game:
  def getBoard(self):
    return self.board # a dict, with the state of the board
  def move(self, playerName, x, y):
    self.board[x,y] = playerName

class PlayerPerspective(pb.Perspective):
  def __init__(self, playerName, game):
    self.playerName = playerName
    self.game = game
  def perspective_move(self, x, y):
    self.game.move(self.playerName, x, y)
  def perspective_getBoard(self):
    return self.game.getBoard()
```

The code on the server side creates the PlayerPerspective object, giving it the right playerName and a reference to the Game object. The remote player doesn't get a reference to the Game object, only their own PlayerPerspective, so they don't have an opportunity to lie about their name: it comes from the `.playerName` attribute, not an argument of their remote method call.

Here is a brief example of using a Perspective. Most of the support code is magic for now: we'll explain it later.

> **Note:**
> This example has more support code than you'd actually need. If you only have one Service, then there's probably a one-to-one relationship between your Identities and your Perspectives. If that's the case, you can use a utility method called Perspective.makeIdentity() instead of creating the perspectives and identities in separate steps. This is shorter, but hides some of the details that are useful here to explain what's going on. Again, this will make more sense later.

```
#! /usr/bin/python

from twisted.spread import pb
from twisted.cred.authorizer import DefaultAuthorizer
import twisted.internet.app

class MyPerspective(pb.Perspective):
    def perspective_foo(self, arg):
        print "I am", self.myname, "perspective_foo(",arg,") called on", self

# much of the following is magic
app = twisted.internet.app.Application("pb5server")
```

[13] Actually there is a perspective per user*service, but we'll get into that later

```
auth = DefaultAuthorizer(app)
# create the service, tell it to generate MyPerspective objects when asked
s = pb.Service("myservice", app, auth)
s.perspectiveClass = MyPerspective

#  create a MyPerspective
p1 = s.createPerspective("perspective1")
p1.myname = "p1"
# create an Identity, give it a name and password, and allow it access to
# the MyPerspective we created before
i1 = auth.createIdentity("user1")
i1.setPassword("pass1")
i1.addKeyByString("myservice", "perspective1")
auth.addIdentity(i1)


# start the application
app.listenTCP(8800, pb.BrokerFactory(pb.AuthRoot(auth)))
app.run(save=0)
```

Source listing — *pb5server.py*

```
#! /usr/bin/python

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

def main():
    def1 = pb.connect("localhost", 8800,
                      "user1", "pass1",
                      "myservice", "perspective1",
                      30)
    def1.addCallbacks(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*

Note that once this example has done the method call, you'll have to terminate both ends yourself. Also note that the Perspective's `.attached()` and `.detached()` methods are run when the client connects and disconnects. The base class implementations of these methods just prints a message.

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, each with their own Perspective. We also override `.attached()` and `.detached()`, just to see how they are called.

Note:

The Perspective object is usually expected to outlast the user's connection to it: it is nominally created some time before the user connects, and survives after they disconnect. `.attached()` and `.detached()` are invoked to let the Perspective know when the user has connected and disconnected.

When the client runs `pb.connect` to establish the connection, 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 server-side Perspective's `.attached` method, in the `clientref` 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 game, this would provide a way to tell the clients that someone has made a move, so they can update their game boards. To actually use it, you'd probably want to subclass Perspective and change the .attached method to stash the clientref somewhere, because the default implementation just drops it.

`.attached()` also receives a reference to the `Identity` object that represents the user. (The user has proved, by using a password of some sort, that they are that `Identity`, and then they can access any service/perspective on the Identity's keyring). The method can use that reference to extract more information about the user.

In addition, `.attached()` has the opportunity to return a different Perspective, if it so chooses. You could have all users initially access the same Perspective, but then as they connect (and `.attached()` gets called), give them unique Perspectives based upon their individual Identities. The client will get a reference to whatever `.attached()` returns, so the default case is to 'return self'.

Finally, when the client goes away (i.e., the network connection has been closed), `.detached()` will be called. The Perspective can use this to mark the user as having gone away: this may mean that outgoing messages should be queued in the Perspective until they reconnect, or callers should be given an error message because they messages cannot be delivered, etc. It can also be used to terminate or suspend any sessions the user was participating in. `detached` is called with the same 'clientref' and Identity objects that were given to the original 'attached' call. It will be invoked on the Perspective object that was returned by `.attached()`.

```
#! /usr/bin/python

from twisted.spread import pb
from twisted.cred.authorizer import DefaultAuthorizer
import twisted.internet.app

class MyPerspective(pb.Perspective):
    def attached(self, clientref, identity):
        print "client attached! they are:", identity
        return self
    def detached(self, ref, identity):
        print "client detached! they were:", identity
```

```
    def perspective_foo(self, arg):
        print "I am", self.myname, "perspective_foo(",arg,") called on", self

# much of the following is magic
app = twisted.internet.app.Application("pb6server")
auth = DefaultAuthorizer(app)
# create the service, tell it to generate MyPerspective objects when asked
s = pb.Service("myservice", app, auth)
s.perspectiveClass = MyPerspective

#  create one MyPerspective
p1 = s.createPerspective("perspective1")
p1.myname = "p1"
# create an Identity, give it a name and password, and allow it access to
# the MyPerspective we created before
i1 = auth.createIdentity("user1")
i1.setPassword("pass1")
i1.addKeyByString("myservice", "perspective1")
auth.addIdentity(i1)

#  create another MyPerspective
p2 = s.createPerspective("perspective2")
p2.myname = "p2"
i2 = auth.createIdentity("user2")
i2.setPassword("pass2")
i2.addKeyByString("myservice", "perspective2")
auth.addIdentity(i2)


# start the application
app.listenTCP(8800, pb.BrokerFactory(pb.AuthRoot(auth)))
app.run(save=0)
```

Source listing — *pb6server.py*

```
#! /usr/bin/python

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

def main():
    def1 = pb.connect("localhost", 8800,
                      "user1", "pass1",
                      "myservice", "perspective1",
                      30)
    def1.addCallbacks(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

def main():
    def1 = pb.connect("localhost", 8800,
                      "user2", "pass2",
                      "myservice", "perspective2",
                      30)
    def1.addCallbacks(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 logs into a different Perspective.

5.4.4 Class Overview

Now that we've seen some of the motivation behind the Perspective class, let's start to de-mystify some of the parts labeled "magic" in `pb6server.py`. Here are the major classes involved:

- `Application`: `twisted/internet/app.py`
- `Service`: `twisted/cred/service.py`
- `Authorizer`: `twisted/cred/authorizer.py`
- `Identity`: `twisted/cred/identity.py`

- `Perspective`: `twisted/cred/pb.py`

You've already seen `Application`. It holds the program-wide settings, like which uid/gid it should run under, and contains a list of ports that it should listen on (with a Factory for each one to create Protocol objects). When used for PB, we put a pb.BrokerFactory on the port. The `Application` also holds a list of Services.

A `Service` is, well, a service. A web server would be a `Service`, as would a chat server, or any other kind of server you might choose to run. What's the difference between a `Service` and an `Application`? You can have multiple `Services` in a single `Application`: perhaps both a web-based chat service and an IM server in the same program, that let you exchange messages between the two. Or your program might provide different kinds of interfaces to different classes of users: administrators could get one `Service`, while mere end-users get a less-powerful `Service`.

> **Note:**
> Note that the `Service` is a server of some sort, but that doesn't mean there's a one-to-one relationship between the `Service` and the TCP port that's being listened to. In theory, several different `Services` can hang off the same TCP port. Look at the MultiService class for details.

The `Service` is reponsible for providing `Perspective` objects. More on that later.

The `Authorizer` is a class that provides `Identity` objects. The abstract base class is `twisted.cred.authorizer.Authorizer`, and for simple purposes you can just use `DefaultAuthorizer`, which is a subclass that stores pre-generated Identities in a simple dict (indexed by username). The `Authorizer`'s purpose in life is to implement the `.getIdentityRequest()` method, which takes a user name and (eventually) returns the corresponding `Identity` object.

Each `Identity` object represents a single user, with a username and a password of some sort. Its job is to talk to the as-yet-anonymous remote user and verify that they really are who they claim to be. The default `twisted.cred.authorizer.Identity` class implements MD5-hashed challenge-response password authorization, much like the HTTP MD5-Authentication method: the server sends a random challenge string, the client concatenates a hash of their password with the challenge string, and sends back a hash of the result. At this point the client is said to be "authorized" for access to that `Identity`, and they are given a remote reference to the `Identity` (actually a wrapper around it), giving them all the privileges of that `Identity`.

Those privileges are limited to requesting `Perspectives`. The `Identity` object also has a "keyring", which is a list of (serviceName, perspectiveName) pairs that the corresponding authorized user is allowed to access. Once the user has been authenticated, the `Identity`'s job is to implement `.requestPerspectiveForKey()`, which it does by verifying the "key" exists on the keyring, then asking the matching `Service` to do `.getPerspectiveForIdentity()`.

Finally, the `Perspective` is the subclass of pb.Perspective that implements whatever `perspective_*` methods you wish to expose to an authenticated remote user. It also implements `.attached()` and `.detached()`, which are run when the user connects (actually when they finish the authentication sequence) or disconnects. Each `Perspective` has a name, which is scoped to the `Service` which owns the `Perspective`.

5.4.5 Class Responsibilities

Now that we've gone over the classes and objects involved, let's look at the specific responsibilities of each. Most of these classes are on the hook to implement just one or two particular methods, and the rest of the class is just support code (or the main method has been broken up for ease of subclassing). This section indicates what those main methods are and when they get called.

Authorizer

The `Authorizer` has to provide `Identity` objects (requested by name) by implementing `.getIdentityRequest()`. The `DefaultAuthorizer` class just looks up the name in a dict called `self.identities`, so when you use it, you have to make the Identities ahead of time (using `i = auth.createIdentity()`) and store them in that dict (by handing them to `auth.addIdentity(i)`).

However, you can make a subclass of `Authorizer` with a `.getIdentityRequest` method that behaves differently: your version could look in `/etc/passwd`, or do an SQL database lookup[14], or create new Identities for anyone that asks (with a really secret password like '1234' that the user will probably never change, even if you ask them to). The Identities could be created by your server at startup time and stored in a dict, or they could be pickled and stored in a file until needed (in which case `.getIdentityRequest()` would use the username to find a file, unpickle the contents, and return the resulting `Identity` object), or created brand-new based upon whatever data you want. Any function that returns a Deferred (that will eventually get called back with the `Identity` object) can be used here.

> **Note:**
> For static Identities that are available right away, the Deferred's callback() method is called right away. This is why the interface of `.getIdentityRequest()` specifies that its Deferred is returned unarmed, so that the caller has a chance to actually add a callback to it before the callback gets run. (XXX: check, I think armed/unarmed is an outdated concept)

Identity

The `Identity` object thus returned has two responsibilities. The first is to authenticate the user, because so far they are unverified: they have claimed to be somebody (by giving a username to the Authorizer), but have not yet proved that claim. It does this by implementing `.verifyPassword`, which is called by IdentityWrapper (described later) as part of the challenge-response sequence. If the password is valid, `.verifyPassword` should return a Deferred and run its callback. If the password is wrong, the Deferred should have the error-back run instead.

The second responsibility is to provide `Perspective` objects to users who are allowed to access them. The authenticated user gives a service name and a perspective name, and `.requestPerspectiveForKey()` is invoked to retrieve the given `Perspective`. The `Identity` is the one who decides which services/perspectives the user is allowed to access. Unless you override it in a subclass, the default implementation uses a simple dict called `.keyring`, which has keys that are (servicename, perspectivename) pairs. If the requested name pair is in the keyring, access is allowed, and the `Identity` will proceed to ask the `Service` to give back the specified `Perspective` to the user. `.requestPerspectiveForKey()` is required to return a Deferred, which will eventually be called back with a `Perspective` object, or error-backed with a `Failure` object if they were not allowed access.

XXX: explain perspective names being scoped to services better

You could subclass `Identity` to change the behavior of either of these, but chances are you won't bother. The only reason to change `.verifyPassword()` would be to replace it with some kind of public-key verification scheme, but that would require changes to `pb.IdentityWrapper` too, as well as significant changes on the client side. Any changes you might want to make to `.requestPerspectiveForKey()` are probably more appropriate to put in the Service's `.getPerspectiveForIdentity` method instead. The Identity simply passes all requests for Perspectives off to the Service.

The default `Identity` objects are created with a username and password, and a "keyring" of valid service/perspective name pairs. They are children of an `Authorizer` object. The best way to create them is to have the

[14] See twisted.enterprise.dbcred for a module that does exactly that.

Authorizer do it for you, then fill in the details, by doing the following:

```
i = auth.createIdentity("username")
i.setPassword("password")
i.addKeyByString("service", "perspective")
auth.addIdentity(i)
```

Service

The Service object's job is to provide Perspective instances, by implementing .getPerspectiveFor Identity(). This function takes a Perspective name, and is expected to return a Deferred which will (eventually) be called back with an instance of Perspective (or a subclass).

The default implementation (in twisted.spread.pb.Service) retrieves static pre-generated Perspectives from a dict (indexed by perspective name), much like DefaultAuthorizer does with Identities. And like Authorizer, it is very useful to subclass pb.Service to change the way .getPerspective ForIdentity() works: to create Perspectives out of persistent data or database lookups, to set extra attributes in the Perspective, etc.

When using the default implementation, you have to create the Perspectives at startup time. Each Service object has an attribute named .perspectiveClass, which helps it to create the Perspective objects for you. You do this by running p = svc.createPerspective("perspective_name").

You should use .createPerspective() rather than running the constructor of your Perspective-subclass by hand, because the Perspective object needs a pointer to its parent Service object, and the Service needs to have a list of all the Perspectives that it contains.

5.4.6 How that example worked

Ok, so that's what everything is supposed to do. Now you can walk through the previous example and see what was going on: we created a subclass called MyPerspective, made a DefaultAuthorizer and added it to the Application, created a Service and told it to make MyPerspectives, used .createPerspective() to build a few, for each one we made an Identity (with a username and password), and allowed that Identity to access a single MyPerspective by adding it to the keyring. We added the Identity objects to the Authorizer, and then glued the authorizer to the pb.BrokerFactory.

How did that last bit of magic glue work? I won't tell you here, because it isn't very useful to override it, but you effectively hang an Authorizer off of a TCP port. The combination of the object and methods exported by the pb. AuthRoot object works together with the code inside the pb.connect() function to implement both sides of the challenge-response sequence. When you (as the client) use pb.connect() to get to a given host/port, you end up talking to a single Authorizer. The username/password you give get matched against the Identities provided by that authorizer, and then the servicename/perspectivename you give are matched against the ones authorized by the Identity (in its .keyring attribute). You eventually get back a remote reference to a Perspective provided by the Service that you named.

> **Note:**
> Here is how the magic glue code works:
>
> ```
> app.listenTCP(8800, pb.BrokerFactory(pb.AuthRoot(auth)))
> ```
>
> pb.AuthRoot() provides objects that are subclassed from pb.Root, so as we saw in the first example, they can be served up by pb.BrokerFactory(). AuthRoot happens to use the .root

Object hook described earlier to serve up an AuthServ object, which wraps the Authorizer and offers a method called .remote_username, which is called by the client to declare which Identity it claims to be. That method starts the challenge-response sequence.

5.4.7 Code Walkthrough: pb.connect()

So, now that you've seen the complete sequence, it's time for a code walkthrough. This will give you a chance to see the places where you might write subclasses to implement different behaviors. We will look at what happens when pb6client1.py meets pb6server.py. We tune in just as the client has run the pb.connect() call.

The client-side code can be summarized by the following sequence of function calls, all implemented in twisted/spread/pb.py . pb.connect() calls getObjectAt() directly, after that each step is executed as a callback when the previous step completes.

```
getObjectAt(host,port,timeout)
logIn(): authServRef.callRemote('username', username)
_cbLogInRespond(): challenger.callRemote('respond', f[challenge,password])
_cbLogInResponded(): identity.callRemote('attach', servicename,
                                         perspectivename, client)
usercallback(perspective)
```

The client does getObjectAt() to connect to the given host and port, and retrieve the object named root. On the server side, the BrokerFactory accepts the connection, asks the pb.AuthRoot object for its .root Object(), getting an AuthServ object (containing both the authorizer and the Broker protocol object). It gives a remote reference to that AuthServ out to the client.

Now the client invokes the '.remote_username' method on that AuthServ. The AuthServ asks the Authorizer to .getIdentityRequest(): this retrieves (or creates) the Identity. When that finishes, it asks the Identity to create a random challenge (usually just a random string). The client is given back both the challenge and a reference to a new AuthChallenger object which will only accept a response that matches that exact challenge.

The client does its part of the MD5 challenge-response protocol and sends the response to the Auth Challenger's .remote_response() method. The AuthChallenger verifies the response: if it is valid then it gives back a reference to an IdentityWrapper, which contains an internal reference to the Identity that we now know matches the user at the other end of the connection.

The client then invokes the .remote_attach method on that IdentityWrapper, passing in a service-Name, perspectiveName, and remoteRef. The wrapper asks the Identity to get a perspective using identity. requestPerspectiveForKey, which does the "is this user allowed to get this service/perspective" check by looking at the tuples on its .keyring, and if that is allowed then it gets the Service (by giving serviceName to the authorizer), then asks the Service to provide the perspective (with svc.getPerspectiveForIdentity). The default Service will ignore the identity object and just look for Perspectives by perspectiveName. The Service looks up or creates the Perspective and returns it. The .remote_attach method runs the Perspective's .attached method (although there are some intermediate steps, in IdentityWrapper._attached, to make sure .detached will eventually be run, and the Perspective's .brokerAttached method is executed to give it a chance to return some other Perspective instead). Finally a remote reference to the Perspective is returned to the client.

The client gives the Perspective reference to the callback that was attached to the Deferred that pb. connect() returned, which brings us back up to the code visible in pb6client1.py.

5.4.8 A Larger Example

Now it's time to look more closely at the Go server described before.

To simplify the example, we will build a server that handles just a single game. There are a variety of players who can participate in the game, named Alice, Bob, etc (the usual suspects). Two of them log in, choose sides, and begin to make moves.

We assume that the rules of the game are encapsulated into a `GoGame` object, so we can focus on the code that handles the remote players.

XXX: finish this section

That's the end of the tour. If you have any questions, the folks at the welcome office will be more than happy to help. Don't forget to stop at the gift store on your way out, and have a really nice day. Buh-bye now!

Chapter 6

Web Applications

6.1 Webizing your application with DOMTemplate

6.1.1 Intro

Most templating systems provide commands that you embed in the HTML to repeat elements or include fragments from other files. This works fairly well for simple constructs; however, as soon as the programmer wants to make the logic even slightly more complicated, the templating system must be bent in ways it was never meant to be used.

The theory behind `DOMTemplate` is that Python code, rather than special syntax in the HTML template, should be used to manipulate the structure of the HTML. `DOMTemplate` uses the Document Object Model[1] (DOM), a W3[2] standard tree-based representation of an HTML document. The DOM provides an API that allows you to traverse nodes in the tree, examine their attributes, move, add, and delete them. For more information on using DOM in Python, see The xml.dom module documentation[3].

6.1.2 Before we begin

You'll need to start a plain Twisted Web server.

```
mktap web --path ~/public_html
twistd -f web.tap
```

For more detailed information on this, see the guide to Installing and Using Twisted.Web (page 22).

Once you do this you should be able to throw any files in `~/public_html`, and they'll be served on localhost:8080. Twisted Web also supports a number of special script types; the one we'll be using here is an `.rpy`, or Resource Script (page 22).

A Resource Script (page 22) is simply a python file ending with the extension `.rpy`, which is required to create an instance of a (subclass of a) `twisted.web.resource.Resource`. The `Resource` subclass we'll be using in this example is, of course, `DOMTemplate`.

Make sure the `TwistedQuotes` directory is on your `PYTHONPATH`, put `webquotes.rpy` and `WebQuotes.html` in your `~/public_html` directory, and you are ready to go.

[1] http://www.w3.org/DOM/

[2] http://www.w3.org/

[3] http://python.org/doc/lib/module-xml.dom.html

6.1.3 Getting To It

There are three files involved in this example; `webquoteresource.py`, `WebQuotes.html`, and `webquotes.rpy`. `webquoteresource.py` is a normal python module and contains the class definition which will be used, a `DOMTemplate` subclass. `WebQuotes.html` is placed in the web directory and is looked up at runtime by the `DOMTemplate` machinery. It is converted into a DOM tree which is iterated during page rendering. Finally, `webquotes.rpy` is placed in the web directory; each time the URL is visited, the file is executed; it `imports` `webquoteresource.QuoteResource` and instantiates it. This instance is asked to render the page.

webquoteresource.py

A `DOMTemplate` subclass must do two things: specify a template, and provide methods to handle specific nodes in the template. The first simply requires either a `template` attribute, which should be a string, or a `templateFile` attribute, which should be a file name, specifying the XHTML template. To accomplish the second, we define methods with the prefix `factory_` in our subclass. When the template is rendered, `DOMTemplate` will look for the `view` attribute on any HTML node. If one is found, the corresponding factory method will be called to handle the node.

```
from twisted.web.woven import template
from twisted.web import domhelpers #helpers for munging the DOM


from TwistedQuotes import quoters

class QuoteResource(template.DOMTemplate):
    """I am a DOMTemplate that displays a fancy quote page."""

    # The template; this must be valid XML (parsable by Python's DOM
    # implementation)
    templateFile = "WebQuotes.xhtml"

    def __init__(self, filenames):
        template.DOMTemplate.__init__(self)
        self.quoter = quoters.FortuneQuoter(filenames)

    def factory_getQuote(self, request, node):
        """
        Return a (hopefully amusing) quote.
        """
        domhelpers.clearNode(node)
        node.appendChild(self.d.createTextNode(self.quoter.getQuote()))
        return node

    def factory_getTitle(self, request, node):
        """Quotes Galore!"""
        domhelpers.clearNode(node)
        node.appendChild(self.d.createTextNode("Quotes Galore!"))
```

```
        return node
```

Listing 1: webquoteresource.py: Twisted Quotes Web Resource module — *webquoteresource.py*

WebQuotes.html

In our example template, we insert a `view` attribute onto the `<title>` node with the value `'getTitle'`. We also insert a `view` attribute on a `<h1>` node with the value `'getTitle'`; this shows the ability of `DOMTemplate` to reuse functionality while applying formatting defined by the template to the output. We also insert a `view` attribute onto a `<pre>` node with the value `'getQuote'`. This is where the real action will take place.

When the DOM is iterated and nodes with view attributes are found, `DOMTemplate` will look in the instance's namespace for a method with the corresponding name, prefixed by `factory_`

```
<html>
    <head>
        <title view="getTitle">
            Title will go here
        </title>
        <style>
            .quote {color: green;}
        </style>
    </head>

    <body>

        <h1 view="getTitle">
            Title will go here
        </h1>

        <pre class="quote" view="getQuote">
            Quote will go here.
        </pre>

    </body>
</html>
```

Listing 2: WebQuotes.html: Twisted Quotes Web Template — *WebQuotes.xhtml*

webquote.rpy

Finally, we need an `.rpy` file in the web directory that twisted can find and execute. This is simply a matter of importing our module, instantiating our class, and assigning the instance to a variable named `resource`. Twisted will discover this instance and call `render` on it, causing the `DOMTemplate` to be rendered.

```
# -*- Python -*-

from TwistedQuotes import webquoteresource
```

```
#__file__ is defined to be the name of this file; this is to
#get the sibling file "quotes.txt" which should be in the same directory
import os
quotefile = os.path.join(os.path.split(__file__)[0], "quotes.txt")

#ResourceScript requires us to define 'resource'.
#This resource is used to render the page.
resource = webquoteresource.QuoteResource([quotefile])
```

Listing 3: webquote.rpy: Twisted Quotes Web Application — *webquote.rpy*

See it in action![4]

6.2 Developing Componentized Web Applications using Woven, the Web Object Visualization Environment

6.2.1 Introduction

DOMTemplate solves the problem of separating logic from presentation, and allows the template manipulation logic to be expressed in Python code form using the DOM API. However, the DOM API is too low level and it quickly becomes tedious to use to build complicated HTML structures.

Twisted's solution is to provide a Model-View-Controller based component framework, which allows you to construct complex HTML "Views" out of many small interacting components, or `woven.widgets`.

Instead of manipulating DOM objects which represent low-level HTML Nodes, you construct and compose the model data that your page will be based on, and specify views which will be responsible for formatting the model data as HTML. Using widgets defined in twisted.web.woven.widgets, and higher-level widgets that you define yourself for an application-specific purpose, python data structures such as strings, integers, lists, dicts, and custom subclasses of `woven.model.Model`, can be adapted implicitly or explicitly to subclasses of `woven.view.View` (such as subclasses of `woven.widgets.Widget`) for display in HTML.

6.2.2 Model-View-Controller

Model View Controller is a development strategy which involves breaking up program logic into three separate domains: Model objects, whose job it is to contain/produce data; View objects, whose job it is to present this data to the user; and Controller objects, whose job it is to handle events such as "user input from a form" and "fetch URL" and update the model with the user's desired changes. When the controller finishes updating the model, it tells the model to notify all views that the model has changed so they may rerender themselves accordingly.

Woven's implementation of MVC uses `twisted.python.components`, the interface and component registry, to loosely couple the interacting objects.

[4] http://twistedmatrix.com/documents/howto/listings/TwistedQuotes/webquote.rpy

6.2.3 Hello World with Page

Let's start with the canonical Hello World example. We will use an instance of `woven.page.Page` as our `IResource` implementor. `IResource` describes the interface that objects are required to implement in order to publish themselves over the web in twisted.web. We will be instanciating a `Page` instance in an .rpy script. An rpy is like a CGI script – each time you visit the script, it is executed. However, an rpy script is merely responsible for instanciating a `Resource` object to handle the request and assigning it to a variable named "resource". This `Resource` object will then be called upon to render the request.

```
from twisted.web.woven import page

resource = page.Page("Hello, world!", templateFile = "HelloWorld.html")
```

Listing 1: HelloWorld.rpy: Hello World Resource Script — *HelloWorld.rpy*

Next, let's take a look at the HTML template woven will look up to render this request into HTML. Woven defines three special attributes, `model=`, `view=`, and `controller=`, which it uses to decide which python code to invoke while rendering the page.

```
<html>
    <head>
        <title model=".">Title will go here.</title>
    </head>
    <body>
        <h3 model=".">Content will go here.</h3>
    </body>
</html>
```

Listing 2: HelloWorld.xhtml: Hello World Web Template — *HelloWorld.html*

In the template, we have simply indicated that woven should replace certain nodes with the results of rendering a widget on the current model. The syntax `model="."` indicates that woven should use the current model no matter what its name, similar to filesystem syntax. Since we aren't explicitly stating *which* view widget should render the model with a `view=` attribute, an `IView` adapter is looked up from the global registry implicitly. In this case, the model is a string, so an instance of `widgets.Text` is constructed which converts the python string into a DOM text node and inserts it into the DOM.

Next, let's look at an example of rendering a page with a more complicated model. We're going to make several pieces of data available to the template under different names. We will refer to these pieces of data as "Submodels", since they are contained in a `Model` instance.

```
from twisted.web.woven import model, page

# The AttributeModel sets submodels as attributes of itself.
# May not be secure theoretically, but we're using it for simple purposes here.
model = model.AttributeModel()

model.setSubmodel("greeting", "Hello, world!")
model.setSubmodel("anInt", 5465465)
```

```
model.setSubmodel("aList", ['fred', 'bob', 'alice', 'joe'])
model.setSubmodel("aDict", {'some': 'stuff', 'goes': 'here'})

resource = page.Page(model, templateFile="HelloWorld2.html")
```

Listing 3: HelloWorld2.rpy: Setting up submodels with a resource script — *HelloWorld2.rpy*

We need a place to gather all the Model data together so the View has access to it. `woven.model.AttributeModel` is a good container to place other models in, and to do so we simply call setSubmodel. Notice that setSubmodel takes a key and a value, the name the submodel will be available as to the template, and the actual submodel data.

Woven comes with various widgets which are registered as `IView` implementors for the basic python types (strings, lists, and dictionaries) which are very useful. Most of the time, you can simply prepare the data for rendering by converting it into strings and lists using an rpy or custom Model subclass, and then referencing these strings and lists in your template.

This time, in our HTML template, we're going to have to be a little more explicit when specifying view widgets to render the model data. It's generally a good idea to always explicitly state the name of the view widget you want to handle a node; but it's convenient that you don't have to, for example if you're rendering a custom model/view pair where it doesn't make sense to use any other view widgets to render a given model instance.

```
<html>
    <head>
        <title model="greeting" view="Text">Title here</title>
    </head>
    <body>
        <h3 model="greeting" view="Text">Greeting here.</h3>

        You are visitor <span model="anInt" view="Text">0</span>! Not really,
        though.

        <ol model="aList" view="List">
            <li listItemOf="aList" view="Text">List item here.</li>
        </ol>

        <span model="aList" view="List">
            <div listItemOf="aList" view="Text" style="background-color: blue">
                List item here.
            </div>
            <div listItemOf="aList" view="Text" style="background-color: green">
                List item here.
            </div>
        </span>

        <p>
            <span model="aDict" view="KeyedList">
                <div keyedListItemOf="aDict" view="Text"
                     style="background-color: blue">List item here.</div>
                <div keyedListItemOf="aDict" view="Text"
```

```
                    style="background-color: green">List item here.</div>
            </span>
        </p>

        <p model="aList" view="None">
            <div model="3">Last item</div>
            <div model="0">First item</div>
        </p>

        <p model="aDict" view="None">
            <div model="some">A dict value here</div>
            <div model="goes">Another dict value here</div>
        </p>

    </body>
</html>
```

Listing 4: HelloWorld2.html: Explicitly stating view widget names in the template — *HelloWorld2.html*

As you can see from the template, the `List` widget requires specially tagged nodes inside of it's node in order to operate properly. These nodes are called "pattern" nodes, and each widget can choose to require certain patterns, or look for certain optional patterns, during the course of rendering itself. In the list widget's case, it looks for the pattern "listItem" and makes one copy of it for each element in the list it is rendering. There are two ways to specify a patten node:

- By putting a `pattern="patternName"` attribute on a node
- By putting a `patternName + 'Of'="modelName"` attribute on a node

This is easier to show by example than to explain; look at the template for an example of the second usage.

Look at the documentation for each individual widget to see what patterns a `Widget` supports. The `List` widget is particularly useful; it supports the following patterns:

- listItem
- listHeader
- listFooter
- emptyList

Notice a few things about this template. First, we are explicitly stating the view widget we wish to render each node with a `view=` attribute. All these view widgets are defined in `woven/widgets.py`, the default woven widgets library. You can also create your own widget libraries for your views, as well as defining subwidget names on more complicated views that are only valid within that views's HTML node. Procedures for doing so will be described in later HOWTOs.

You now know how to create a woven HTML template, and how to populate this template with data in the form of simple python data types. However, often you will wish to render a dynamic data source, such as a database, or a complex data source such as a python object. One way to render this data over the web is to create a class which implements `IModel`, the interface woven uses to expose data to view widgets.

6.2.4 Implementing IModel

The `IModel` interface is documented in `twisted.web.woven.interfaces`. It describes the interfaces Models must implement in order to play well with the rest of the woven MVC framework. 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`.

```
# wovenquotes

from twisted.web.woven import model, input
from TwistedQuotes import quoters


class MQuote(model.Model):
    """A class which implements IModel for a FortuneQuoter instance for a given
    filename.
    """
    def __init__(self, filename):
        model.Model.__init__(self)
        self._filename = filename
        self._quoter = quoters.FortuneQuoter([filename])

    def getData(self):
        """Get a random quote from the quotefile.
        """
        return self._quoter.getQuote()

    def setData(self, data):
        """Add a new quote to the quotefile.
        """
        file = open(self._filename, 'a')
        file.write('\n%\n'  + data)
```

Listing 5: wovenquotes.py: Implementing IModel to provide custom Model behavior — *wovenquotes.py*

We have created a simple Model which wraps a quoter that was created in a previous HOWTO. The constructor stores the filename and creates a new FortuneQuoter instance.

Implementing `getData` is as simple as delegating to our FortuneQuoter instance. `getQuote` returns a string. In the template, we will specify that the `Text` widget should render the data returned by the quote model, so the quote shows up in the template. But first, we need to get an instance of MQuote into the model namespace, using an rpy:

```
# -*- Python -*-

from twisted.web.woven import model, page
from TwistedQuotes import wovenquotes
```

```
#__file__ is defined to be the name of this file; this is to
#get the sibling file "quotes.txt" which should be in the same directory
import os
quotefile = os.path.join(os.path.split(__file__)[0], "quotes.txt")

# ResourceScript requires us to define 'resource'. This resource is used
# to render the page.

# We're passing a dictionary of model data the template can render.
# A static title and an instance of our custom Model subclass MQuote.

model = {'quote': wovenquotes.MQuote(quotefile),
                        'title': "Woven Quotes!"}

resource = page.Page(model, templateFile="WovenQuotes.xhtml")
```

Listing 6: wovenquotes.rpy: Tying together a custom Model subclass and an html template with an rpy — *wovenquotes.rpy*

This time, instead of using an instance of AttributeModel as our main model namespace, we have chosen to simply use a dictionary. Since a dictionary doesn't implement `IModel`, an attribute lookup occurs which wraps the dictionary in an instance of `DictionaryModel`, which does implement `IModel`. Then all of the dictionary keys will be available as submodel names.

```
<html>
    <head>
        <title model="title">
            Quotes Galore!
        </title>
        <style>
            .quote {color: green;}
        </style>
    </head>

    <body>

        <h1 model="title">
            Quotes Galore!
        </h1>

        <pre view="Text" model="quote">
            Quote will go here.
        </pre>

        <form action="">
            <input type="text" name="quote" model="quote"
           controller="Anything" />
```

```
            <input type="submit" />
        </form>
        <a href="wovenquotes.rpy">Refresh</a>

    </body>
</html>
```

Listing 7: WovenQuotes.xhtml: WovenQuotes Template — *WovenQuotes.xhtml*

Woven templates are designed to minimize the amount of logic contained in the HTML template. A woven template is a collection of HTML nodes that are tagged with various strings which woven uses to locate python components that are then responsible for producing the final output. In this case, we joined the output of our `Model` subclass' `getData` (a string) with the `Text` widget (which knows how to render strings into DOM).

The same theory is used for input handling by `Controllers`. In the template, we have placed the following input node:

```
<input type="text" name="quote" model="quote" controller="Anything" />
```

When woven encounters this node, it will look up the submodel name "quote", and will get an instance of MQuote. It will then look up the controller name "`Anything`". `Anything` is an `InputHandler` (a specialized type of controller designed to handle web request input) defined in `twisted.web.woven.input`. Similarly to `woven.widgets`, `woven.input` is used as a default controller namespace when searching for controller names.

The `AnythingInputHandler` looks in the request arguments for data with the same name as its model name. In this case, the model name is "quote", and the input node also has the name "quote". When the form is submitted, the value of the text field will be available in the request arguments as "quote", and the `AnythingInputHandler` will find it. The `AnythingInputHandler` then immediately calls `model.setData(value)` since it does no input validation.

`Under construction.`

Next up: Implementing custom view logic with wvupdate, and creating and using reusable `Widget` and `InputHandler` subclasses with wvfactory and wcfactory.

6.3 Introducing Twisted Web Widgets

> **Note:**
> `twisted.web.widgets` is being gradually deprecated in favour of Woven (page 136). See the module docstring for details.

6.3.1 Introduction

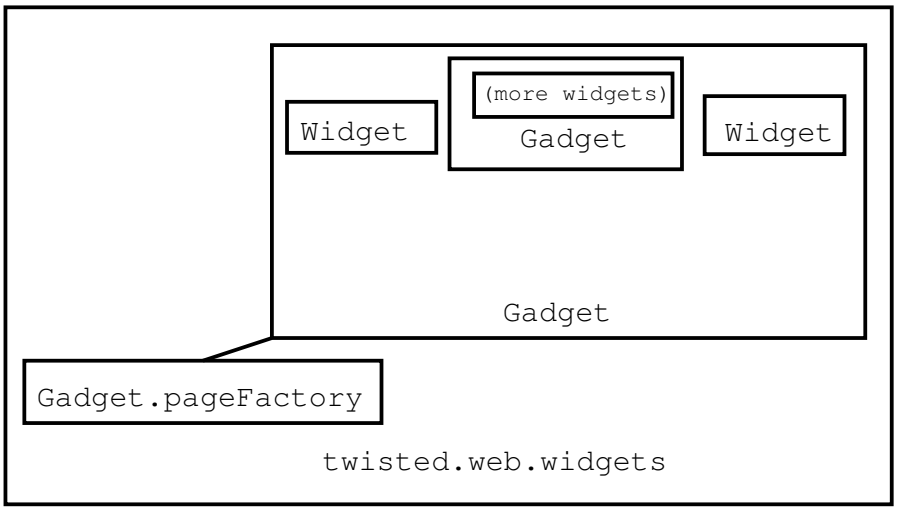
This is more of a simple description of all the classes, plus the common pitfalls of coding in Web Widgets. Oh well.

6.3.2 Example Code

Chris Armstrong has made some example (contrived) Widgets code available, at http://twistedmatrix.com/users/carmstro.twistd/files/Example.tar.gz. Unpack it into your ~/TwistedPlugins / directory and run `twistd -g Example` somewhere to start the server on localhost:8080. Please read the code

(comments) before getting confused about the odd behavior of the example server – note that you are *supposed* to get a `No Resource` error on the root URL (http://localhost:8080/) when you first load it up; the code explains this.

6.3.3 The Diagram



6.3.4 The Classes

Gadget

A collection of widgets, like a "directory" of HTML files. You add widgets to it with `self.putWidget("name", WidgetInstance())`. This widget will be rendered inside the Gadget-local page. Also, if you make a Gadget that is also a subclass of "Widget", then whenever the "index" (http://foo.com/foo/, "foo" being the Gadget/Widget resource) is requested, the object will be rendered as a Widget inside of the Gadget-local page factory. The Gadget-local page factory is the '`pageFactory`' attribute of the gadget, which should be a class that takes a widget in it's constructor, and displays that Widget in some form. So in your `__init__` method for your Gadget subclass, do `self.pageFactory = SomeWidgetPageSubclass` (see "WidgetPage" below) (note that it is *not* an instance, but the actual class object).

Widget

A Widget is simply something that is renderable, through its `display()` method. This method is expected to return a list of HTML strings. (it can also contain instances of defer.Deferred – but this is another story).

Presentation

This is a special Widget that already has a display() method, which renders some objects through a template. You override the special 'template' variable, which is a string with interpolated python expressions. It should look something like:

```
template = '''\
    <html>
    <head><title>%%%%self.title%%%%</title></head>
    <body>%%%%self.getContent(request)%%%%
    </body></html>
'''
```

As you can see, Python expressions are denoted with surrounding sets of 4 %s. The expressions are evaluated in a special namespace with only 'self' and 'request' in it.

WidgetPage

A WidgetPage is a special Page/Presentation combination that allows you to pass a Widget object to its constructor. The most common use of this class is for subclassing; you should have a subclass that defines a custom 'template' attribute. WidgetPage stores the widget you pass to it in it's 'widget' attribute, so remember that whenever you're making a customized template, use %%%%self.widget%%%% to access it (see "Common Pitfalls: WidgetPage" below).

6.3.5 Common Pitfalls

WidgetPage

If you have a subclass of widgets.WidgetPage , make sure your template accesses the widget it's displaying with the 'self.widget' object. For example, if you want to get the title from the current widget you're displaying:

```
template = '''\
<html>
<head><title>%%%%self.widget.title%%%%</title>
</head></html>
'''
```

instead of:

```
template = '''\
<html>
<head><title>%%%%title%%%%</title></head>
</html>
'''
```

Adding Widgets to a Gadget

I had some code like this in one of my Gadgets: self.putWidget("Foo", widgets.TitleBox(My Widget())). Later whenever trying to access this widget I got this traceback (word wrapped for readability):

```
Traceback evaluating code in twisted.words.webwords.Page:Traceback
(most recent call last):
  File "/usr/lib/python2.1/site-packages/twisted/web/widgets.py",
  line 86, in display
```

```
    x = eval(elem, namespace, namespace)
  File "<string>", line 0, in ?
AttributeError: TitleBox instance has no attribute 'getHeader'
```

Now remember, widgets that you add to a gadget with putWidget are rendered with self.pageFactory like so: `self.pageFactory(theChildWidget)`. The problem is, theChildWidget in this case was actually TitleBox! and of course, TitleBox doesn't follow our template's protocol of having a '`getHeader`' method. So, the lesson is: do not wrap your real widgets with other widgets when adding to a Gadget: do formatting either in a) the template or b) the widget's `display()` method.

Return values of display()

If you ever get this traceback (word wrapped for readability):

```
web.Server Traceback

Traceback (most recent call last):
  File "/home/punck/cvs/Twisted/twisted/web/server.py", line 215, in process
    body = resrc.render(self)
  File "/usr/lib/python2.1/site-packages/twisted/web/widgets.py", line 408,
  in render
    displayed = self.display(request)
  File "/usr/lib/python2.1/site-packages/twisted/web/widgets.py", line 97,
  in display
    tm.extend(val)
AttributeError: TitleBox instance has no attribute '__len__'
```

It's because you tried to put a widget in the list that display() returns! For now, just tack on `.display(request)` to all the widgets you want to return in that list.

6.4 Light Weight Templating With Resource Templates

6.4.1 Overview

While Twisted supports solution like Woven (page 136) 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 and ResourceScript using Quixote's PTL (Python Templating Language).

ResourceTemplates need Quixote installed. In Debian[5], 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.

6.4.2 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:

[5] http://www.debian.org

```
% 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.

6.4.3 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[6]. 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):
        """\
        <html>
        <head><title>Quotes Galore</title></head>

        <body><h1>Quotes</h1>"""
        quoter.getQuote()
        "</body></html>"


resource = QuoteResource()
```

Resource Template for Quotes — *webquote.rtl*

6.5 Creating XML-RPC Servers and Clients with Twisted

6.5.1 Introduction

XML-RPC[7] is a simple request/reply protocol that runs over HTTP. It is simple, easy to implement and supported by most computer languages. Twisted's XML-RPC support uses the xmlrpclib library for parsing - it's included with

[6] http://www.mems-exchange.org/software/quixote/doc/PTL.html
[7] http://www.xmlrpc.com

Python 2.2, but can be downloaded for Python 2.1 from Pythonware[8].

6.5.2 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. They can also return Failure instances to indicate an error has occured, 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 72) 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)`. You can run it directly or with `twistd -y script.py`

```
from twisted.web import xmlrpc, server

class Example(xmlrpc.XMLRPC):
    """An example object to be published."""

    def xmlrpc_echo(self, x):
        """Return all passed args."""
        return x

    def xmlrpc_add(self, a, b):
        """Return sum of arguments."""
        return a + b


def main():
    from twisted.internet.app import Application
    app = Application("xmlrpc")
    r = Example()
    app.listenTCP(7080, server.Site(r))
    return app

application = main()

if __name__ == '__main__':
    application.run(save=0)
```

After we run this command, we can connect with a client and send commands to the server:

```
>>> import xmlrpclib
```

[8] http://www.pythonware.com/products/xmlrpc/

```
>>> 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*

6.5.3 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 two methods, 'add' and 'echo'."""

    def soap_quote(self):
        return quoter.getQuote()

resource = Quoter()
```

Source listing — *soap.rpy*

6.5.4 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 `callMethod`.

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 `callMethod` 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 cauled. 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]
```

Chapter 7

Dot Products

7.1 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.

Chapter 8

Working on the Twisted Code Base

8.1 Twisted Coding Standard

8.1.1 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*.

8.1.2 Testing

Unit tests are written using the PyUnit framework. Many examples are in the `twisted.test` package, and all tests should be integrated through the main test suite builder in `twisted.test.test_all`.

Acceptance tests are all automated by the bin/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 run `admin/generate-domdocs` and check the output is all right. Also, it is useful to run `bin/hlint` on any howto you may have modified. There should never be any warnings – if you're having problems, you can always explicitly silence warnings by using `hlint="off"` (though please do this sparingly). If you modify the quotes file, run `strfile(1)` to check you did not harm it. In particular, if the "shortest string" is too short, you might have trailing empty lines.

8.1.3 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.

8.1.4 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 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*.

8.1.5 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`.

8.1.6 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.
```

8.1.7 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. 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

[1] http://epydoc.sourceforge.net/epytext.html

```
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.

8.1.8 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.

8.1.9 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.

8.1.10 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.

8.1.11 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.

8.1.12 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.

8.1.13 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.

8.1.14 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.

8.1.15 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.

8.1.16 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.

8.2 HTML Documentation Standard for Twisted

8.2.1 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.

8.2.2 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(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(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(self, request):
    ...         return "Hello, world!"
    ...
    &gt;&gt;&gt; MyResource().render(None)
    "Hello, world!"
    </pre>
```

```
>>> from twisted.web import resource
>>> class MyResource(resource.Resource):
...     def render(self, request):
...         return "Hello, world!"
...
>>> MyResource().render(None)
"Hello, world!"
```

"shell"

```
    <pre class="shell">
    $ mktap web --path /var/www
    </pre>
```

```
$ mktap web --path /var/www
```

8.2.3 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.)

8.2.4 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.

8.2.5 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>`

8.2.6 Tag Case

All tags will be done in lower-case. XHTML demands this, and so do I. :-)

8.2.7 Footnotes

Footnotes are enclosed inside `<span class="footnote"></span>`. They must not contain any markup.

8.2.8 Suggestions

Use `hlint` to check your documentation is not broken. `hlint` will never change your HTML, but it will complain if it doesn't like it.

Don't use tables for formatting. 'nuff said.

8.3 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.

8.3.1 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.

8.3.2 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.

"Note: I haven't yet fully figured this out myself. Anyone else with more experience want to enlighten us? – Kevin"

8.3.3 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)
```

[2] http://c2.com/cgi/wiki?ExtremeProgramming

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.

8.3.4 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 161) 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 copy of the PyUnit testing framework. That just means that when you import the unittest module, you will `from pyunit import unittest` instead of the standard `import unittest`.

(Now that unittest has been in the Python standard library for several releases, I'm even less convinced of this argument than before, but doesn't add more than a second or two to download time, so I don't worry it too much.)

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.

8.3.5 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].

[3] http://www.xprogramming.com/xpmag/expUnitTestsAt100.htm
[4] http://www.cvshome.org/docs/manual/cvs_5.html
[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

- Ron Jeffries writes about the Unit Test[15] in the Extreme Programming practices of C3[16].
- PyUnit's homepage[17].
- twisted.test[18]'s inline documentation.
- The twisted/test directory[19] in CVS.

[15] http://www.xprogramming.com/Practices/PracUnitTest.html
[16] http://www.xprogramming.com/Practices/xpractices.htm
[17] http://pyunit.sourceforge.net
[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

Chapter 9

Manual Pages

9.1 COIL.1

9.1.1 NAME

coil - configure twisted TAP files

9.1.2 SYNOPSIS

coil [–new=*name*] <*file.tap*>
coil –help

9.1.3 DESCRIPTION

Once you've launched coil, point your browser at http://localhost:9080 to configure the TAP file, and when done hit Ctrl-C to shutdown and save the changes.

–help Print out a usage message to standard output.

***-n, –new*<*name*>** Create a new twisted Application.

***-p, –port*<*port*>** Run the coil web server on <port> (defaults to 9080)

-l, –localhost Bind only to localhost, instead of to all interfaces, thus only letting local users access coil.

9.1.4 AUTHOR

Written by Itamar Shtull-Trauring, based on coil's help messages

9.1.5 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

9.1.6 COPYRIGHT

9.1.7 SEE ALSO

twistd(1), mktap(1)

9.2 CONCH.1

9.2.1 NAME

conch - connect to SSH servers

9.2.2 SYNOPSIS

conch [-l *user*] [-i *identity* [-i *identity* ...]] [-c *cipher*] [-m *MAC*] [-p *port*] [-n] [-t] [-T] [-V] [-C] [-N] [-s] [arg [...]]
conch –help

9.2.3 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

9.2.4 DESCRIPTION

Open an SSH connection to specified server, and either run the command given there or open a remote interactive shell.

9.2.5 AUTHOR

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

9.2.6 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

9.2.7 COPYRIGHT

9.2.8 SEE ALSO

ssh(1)

9.3 GENERATELORE.1

9.3.1 NAME

generatelore - generate spiffy HTML from a plain subset of HTML

9.3.2 SYNOPSIS

generatelore [-t *template*] [-a *apidir*] [-d *docsdir*] [-e *ext*] [-u *baseurl*] [file [...]]
generatelore –help

9.3.3 DESCRIPTION

The *–help* prints out a usage message to standard output.

-t*, *–template<**template**> The template file to follow for generating content (default: template.tpl)

-a*, *–apidir<**apidir**> The directory in which a copy of the epydoc-generated api documentation is. If available, will allow for non-full references to modules and classes.

-e*, *–ext<**extension**> The new extension (defaults to xhtml)

-u*, *–baseurl<**url**> URL in which the API documentation is available.

9.3.4 DESCRIPTION

The exact HTML syntax to follow is documented in Twisted's doc-standard.html. If no files are given, all *.html documents in docsdir will be processed.

9.3.5 AUTHOR

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

9.3.6 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

9.3.7 COPYRIGHT

9.3.8 SEE ALSO

html2latex(1)

9.4 HLINT.1

9.4.1 NAME

hlint - check an HTML file for non-compliance with the twisted.lore format

9.4.2 SYNOPSIS

hlint [file [file ...]]

9.4.3 DESCRIPTION

Read each specified file and report errors.

9.4.4 AUTHOR

Written by Moshe Zadka

9.4.5 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

9.4.6 COPYRIGHT

9.4.7 SEE ALSO

mktap(1)

9.5 HTML2LATEX.1

9.5.1 NAME

html2latex - generate LaTeX out of a subset of HTML

9.5.2 SYNOPSIS

html2latex [-d *directory*] [-s] [file [...]]
html2latex –help

9.5.3 DESCRIPTION

The *–help* prints out a usage message to standard output.

***-d*, *–directory*<directory>** Directory relative to which links are resolved. The default is the directory in which the file is.

-s*, *–section Generate a section of LaTeX, not an article

9.5.4 DESCRIPTION

The exact HTML syntax to follow is documented in Twisted's doc-standard.html. If no files are given, all *.html documents in docsdir will be processed.

9.5.5 AUTHOR

Written by Moshe Zadka, based on html2latex's help messages

9.5.6 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

9.5.7 COPYRIGHT

9.5.8 SEE ALSO

generatelore(1)

9.6 IM.1

9.6.1 NAME

im - run Instance Messenger, the Tkinter twisted.words client

9.6.2 SYNOPSIS

im

9.6.3 DESCRIPTION

Run Instance Messenger, the Tkinter twisted.words client

9.6.4 AUTHOR

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

9.6.5 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

9.6.6 COPYRIGHT

9.7 MANHOLE.1

9.7.1 NAME

manhole - Connect to a Twisted Manhole service

9.7.2 SYNOPSIS

manhole

9.7.3 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.

9.7.4 AUTHOR

Written by Chris Armstrong, copied from Moshe Zadka's "faucet" manpage.

9.7.5 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

9.7.6 COPYRIGHT

9.8 MKTAP.1

9.8.1 NAME

mktap - create twisted.servers

9.8.2 SYNOPSIS

mktapapptype [*application_option*]...
mktap –help *apptype*

9.8.3 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.

–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', 'sister', 'ftp', and 'mail'. Each of those support different options.

9.8.4 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*>

9.8.5 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 wether or not a request for 'foo' should return 'foo.ext'. Default is off.

-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.

9.8.6 toc options

-p<***port***> <port> is a number representing which port you want to start the server on.

9.8.7 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)

9.8.8 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>

9.8.9 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>

9.8.10 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

9.8.11 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>

9.8.12 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>

9.8.13 AUTHOR

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

9.8.14 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

9.8.15 COPYRIGHT

9.8.16 SEE ALSO

twistd(1)

9.9 IM.1

9.9.1 NAME

t-im - run Instance Messenger, the GTK+ twisted.words client

9.9.2 SYNOPSIS

t-im

9.9.3 DESCRIPTION

Run Instance Messenger, the GTK+ twisted.words client

9.9.4 AUTHOR

Written by Moshe Zadka, based on t-im's code

9.9.5 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

9.9.6 COPYRIGHT

9.10 TAP2DEB.1

9.10.1 NAME

tap2deb - create Debian packages which wrap .tap files

9.10.2 SYNOPSIS

tap2deb [options]

9.10.3 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*, *–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 metafiles, as well as be used as the id to sign the package.

9.10.4 AUTHOR

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

9.10.5 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

9.10.6 COPYRIGHT

9.10.7 SEE ALSO

mktap(1)

9.11 TAPCONVERT.1

9.11.1 NAME

tapconvert - convert Twisted configurations from one format to another

9.11.2 SYNOPSIS

tapconvert -i *input* -o *output* [-f *input-type*] [-t *output-type*] [-d] [-e]
tapconvert –help

9.11.3 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 'python', 'pickle', 'xml', or 'source'. Default is 'source'.

–decrypt*, *-d Decrypt input.

–encrypt*, *-e Encrypt output.

9.11.4 AUTHOR

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

9.11.5 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

9.11.6 COPYRIGHT

9.11.7 SEE ALSO

mktap(1)

9.12 TRIAL.1

9.12.1 NAME

trial - run unit tests

9.12.2 SYNOPSIS

trial [-vb] [-r *reactor*] [-l *logfile*] [-m *module* [-m *module* ...]] [-p *package* [-p *package* ...]]
trial –help

9.12.3 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). With this option, trial prints a single line for each test. This is especially useful for gauging how long each test takes.

***-m*, *–module*<module>** Module containing test cases.

***-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.

-b*, *–debug Run the tests in the Python debugger.

9.12.4 AUTHOR

Written by Jonathan M. Lange

9.12.5 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

9.12.6 COPYRIGHT

9.13 TWISTD.1

9.13.1 NAME

twistd - run twisted.internet.app.Application pickles

9.13.2 SYNOPSIS

twistd [options]

9.13.3 DESCRIPTION

Read an twisted.internet.app.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 SIGINT signal to the process will drop it into the debugger.

-o*, *–no_save Do not save shutdown state

***-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".

9.13.4 AUTHOR

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

9.13.5 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

9.13.6 COPYRIGHT

9.13.7 SEE ALSO

mktap(1)

9.14 WEBSETROOT.1

9.14.1 NAME

websetroot - set the root of a Twisted web server

9.14.2 SYNOPSIS

websetroot {-f *tapfile* — -y *codefile* — -x *XML* — -s *AOT* } {–pickle *pickle* — –script *script* } [-e] [–port *port*]
websetroot –help

9.14.3 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

9.14.4 AUTHOR

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

9.14.5 REPORTING BUGS

Report bugs to <twisted-python@twistedmatrix.com>.

9.14.6 COPYRIGHT

9.14.7 SEE ALSO

mktap(1)

Chapter 10

Appendix

10.1 The Twisted FAQ

10.1.1 What is "Twisted"?

Please see Twisted[1]

10.1.2 Why should I use Twisted?

See The Twisted Advantage[2]

10.1.3 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.

10.1.4 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

10.1.5 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.

10.1.6 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.

10.1.7 Which parts of Twisted 1.0 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 1.0, *most of twisted.internet is completely stable*, other than:

1. UDP support
2. twisted.internet.win32eventreactor - will be replaced with win32support in future.

But as always, the only accurate way of knowing a module's stability is reading the module's docstrings.

10.1.8 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 58) and Writing Clients (page 62) for more details.

10.1.9 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.

10.1.10 But can Twisted Web do PHP?

Yes. It works out-of-the-box, so long as you've got the standalone php interpreter installed. You might also want to take a look at Woven, Twisted's native web templating system.

10.1.11 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).

10.1.12 Where can I find out how to write Twisted servers?

Try Writing Servers[3].

10.1.13 Twisted is cool, but I need to add more functionality.

Great! Read our the docs, and if you're feeling generous, contribute patches.

10.1.14 I have a patch. How do I maximize the chances the Twisted developers will include it?

Use unified diff. Either use `cvs diff -u` or, better yet, make a clean checkout and use `diff -urN` between them. Make sure your patch applies cleanly. Then, send it to the mailing list inlined and without any word wrapping.

10.1.15 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.

10.1.16 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.

10.1.17 Help!

Ask for help where the Twisted team hangs out[4]

[3] http://www.twistedmatrix.com/documents/howto/servers
[4] http://twistedmatrix.com/services/online-help

10.1.18 I have this cool patch. To whom do I send it?

To the mailing list[5]. If no one picks it up after a days, it's recommended that you add it to the bug tracker[6] so that it doesn't get lost.

10.1.19 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 CVS[7], 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, CVS versions of any important files, Python version, code you wrote or things you did to trigger the bug, etc.) to the mailing list[8]. If no one answers immediately, you should add it to the bug tracker[9].

10.1.20 How do I use twisted.web to do complex things?

See the Twisted.Web Howto (page 22).

10.1.21 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.

10.1.22 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.

10.1.23 Wow the Twisted documentation is nice! I want my docs to look like that too!

Now you can, with `twisted.lore`. See the manual pages for `generatelore` and `html2latex`. For source format documentation, see the documentation standard description (page 157).

10.1.24 twistd won't load my .tap file!

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
```

[5] http://twistedmatrix.com/cgi-bin/mailman/listinfo/twisted-python
[6] http://sourceforge.net/tracker/?group_id=49387&atid=456015
[7] http://twistedmatrix.com/developers/cvs
[8] http://twistedmatrix.com/cgi-bin/mailman/listinfo/twisted-python
[9] http://sourceforge.net/tracker/?group_id=49387&atid=456015

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.

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:

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

### Protocol Implementation

# This is just about the simplest possible protocol

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

### Persistent Application Builder

# This builds a .tap file

if __name__ == '__main__':
    # Since this is persistent, it's important to get the module naming right
    # (If we just used Echo, then it would be __main__.Echo when it attempted
    # to unpickle)
    import echoserv
    from twisted.internet.app import Application
    factory = Factory()
    factory.protocol = echoserv.Echo
```

```
app = Application("echo")
app.listenTCP(8000,factory)
app.listenUDP(8000, factory)
app.save("start")
```

doc/examples/echoserv.py — *echoserv.py*

10.1.25 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 CVS repository.

10.2 Twisted Glossary

`Adapter` An object which wraps another object to conform to a particular Interface (page 190), when some code requests the interface from an object. See `twisted.python.components`.

`Application` A `twisted.internet.app.Application`. There are HOWTOs on creating and manipulating (page 20) them as a system-administrator, as well as using (page 35) them in your code.

`Authorizer` An object responsible for managing Identities (page 190). See `twisted.cred.authorizer`.

`Banana` The low-level data marshalling layer of Twisted Spread (page 191). See `twisted.spread.banana`.

`Broker` A `twisted.spread.pb.Broker`, the object request broker for Twisted Spread (page 191).

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`.

`Componentized` An object that can hold a number of persistent `Adapters`, which we call "components".

`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 62).

Consumer An object that consumes data from a Producer (page 191). See `twisted.internet.interfaces.IConsumer`.

Cred Twisted's authentication API, `twisted.cred`. See Introduction to Twisted Cred (page 15) and Twisted Cred usage (page 120).

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 72) 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 191)". See Introduction to Twisted Enterprise (page 13) and Twisted Enterprise Row Objects (page 47) for more details.

errback A callback attached to a Deferred (page 190) 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 191) instances for incoming or outgoing connections. See Writing Servers (page 58) and Writing Clients (page 62).

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

Identity A Cred (page 189) object that represents a single user with a username and a password of some sort.

im, t-im Abbreviation of "(Twisted) Instance Messenger (page 190)".

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 191) with Twisted Words (page 192). 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`.

Jelly The serialization layer for Twisted Spread (page 191), 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 189), 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 generatelore(1) (page 167) and the Twisted Documentation Standard (page 157).

Manhole A debugging/administration interface to a Twisted application. See Debugging with Manhole (page 26).

Marmalade An XML-based serialisation module. See `twisted.persisted.marmalade`.

Microdom A partial DOM implementation using SUX (page 191). It is simple and pythonic, rather than strictly standards-compliant. See `twisted.web.microdom`.

Names Twisted's DNS server, found in `twisted.names`.

PB Abbreviation of "Perspective Broker (page 191)".

Perspective A Cred (page 189) object; an Identity (page 190)'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 191).

Perspective Broker The high-level object layer of Twisted Spread (page 191), implementing semantics for method calling and object copying, caching, and referencing. See `twisted.spread.pb`.

Producer An object that generates data a chunk at a time, usually to be processed by a Consumer (page 189). 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 58) and Writing Clients (page 62).

PSU There is no PSU.

Reactor The core event-loop of a Twisted application. See Reactor Basics (page 57).

Reality See "Twisted Reality (page 191)"

`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 47).

Service A `twisted.cred.service.Service`. See Twisted Cred usage (page 120) for a description of how they relate to Applications (page 189), Perspectives (page 190) and Identities (page 190).

Spread Twisted Spread[10] is Twisted's remote-object suite. It consists of three layers: Perspective Broker (page 191), Jelly (page 190) and Banana. (page 189) See Writing Applications with Perspective Broker (page 13).

Sturdy A persistent reference manager for PB (page 191). See `twisted.spread.sturdy`.

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 20).

Tendril A bridge between Twisted Words (page 192) and IRC. See `twisted.words.tendril`.

Trial `twisted.trial`, Twisted's unit-testing framework, modelled after pyunit[11].

Twisted Matrix Laboratories The team behind Twisted. http://twistedmatrix.com/.

Twisted Reality In days of old, the Twisted Reality[12] 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 51).

[10] http://twistedmatrix.com/products/spread
[11] http://pyunit.sourceforge.net/
[12] http://twistedmatrix.com/products/reality

Words Twisted Words is a multi-protocol chat server that uses the Perspective Broker (page 191) 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 136).

Zoot Twisted's Gnutella implementation (currently very incomplete). See `twisted.zoot`.

10.3 Banana Protocol Specifications

10.3.1 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.

10.3.2 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).

10.3.3 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 – 0x81 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 – 0x83 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

1234567891234567890 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

10.3.4 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'

10.3.5 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.