

The Twisted Documentation

The Twisted Development Team

January 9, 2007

Contents

1	Introduction	7
1.1	The Vision For Twisted	7
1.2	High-Level Overview of Twisted	7
1.3	Asynchronous Programming with Twisted	8
1.3.1	Introduction to concurrent programming	8
1.3.2	Deferreds	9
1.3.3	The Problem that Deferreds Solve	9
1.3.4	Deferreds - a signal that data is yet to come	10
1.3.5	Conclusion	12
1.4	Overview of Twisted Internet	12
2	Tutorial	13
2.1	Writing Servers	13
2.1.1	Overview	13
2.1.2	Protocols	13
2.1.3	Factories	15
2.2	Writing Clients	17
2.2.1	Overview	17
2.2.2	Protocol	17
2.2.3	Simple, single-use clients	17
2.2.4	ClientFactory	18
2.2.5	A Higher-Level Example: ircLogBot	19
2.3	Setting up the TwistedQuotes application	22
2.3.1	Goal	22
2.3.2	Setting up the TwistedQuotes project directory	22
2.4	Designing Twisted Applications	23
2.4.1	Goals	23
2.4.2	Example of a modular design: TwistedQuotes	23
2.5	Twisted from Scratch, or The Evolution of Finger	25
2.5.1	Introduction	25
2.5.2	Contents	25
2.6	The Evolution of Finger: building a simple finger service	25
2.6.1	Introduction	25
2.6.2	Refuse Connections	25
2.6.3	Do Nothing	26
2.6.4	Drop Connections	26
2.6.5	Read Username, Drop Connections	26
2.6.6	Read Username, Output Error, Drop Connections	27
2.6.7	Output From Empty Factory	27
2.6.8	Output from Non-empty Factory	27
2.6.9	Use Deferreds	28
2.6.10	Run 'finger' Locally	28
2.6.11	Read Status from the Web	29
2.6.12	Use Application	29
2.6.13	twistd	30

2.7	The Evolution of Finger: adding features to the finger service	30
2.7.1	Introduction	30
2.7.2	Setting Message By Local Users	30
2.7.3	Use Services to Make Dependencies Sane	31
2.7.4	Read Status File	32
2.7.5	Announce on Web, Too	33
2.7.6	Announce on IRC, Too	34
2.7.7	Add XML-RPC Support	36
2.8	The Evolution of Finger: cleaning up the finger code	37
2.8.1	Introduction	37
2.8.2	Write Readable Code	37
2.9	The Evolution of Finger: moving to a component based architecture	40
2.9.1	Introduction	40
2.9.2	Write Maintainable Code	40
2.9.3	Advantages of Latest Version	45
2.9.4	Aspect-Oriented Programming	50
2.10	The Evolution of Finger: pluggable backends	50
2.10.1	Introduction	50
2.10.2	Another Back-end	50
2.10.3	Yet Another Back-end: Doing the Standard Thing	55
2.11	The Evolution of Finger: a web frontend	61
2.11.1	Introduction	61
2.12	The Evolution of Finger: Twisted client support using Perspective Broker	65
2.12.1	Introduction	65
2.12.2	Use Perspective Broker	65
2.13	The Evolution of Finger: using a single factory for multiple protocols	71
2.13.1	Introduction	71
2.13.2	Support HTTPS	71
2.14	The Evolution of Finger: a Twisted finger client	77
2.14.1	Introduction	77
2.14.2	Finger Proxy	77
2.15	The Evolution of Finger: making a finger library	79
2.15.1	Introduction	79
2.15.2	Organization	79
2.15.3	Easy Configuration	80
2.16	The Evolution of Finger: configuration and packaging of the finger service	81
2.16.1	Introduction	81
2.16.2	Plugins	82
2.16.3	OS Integration	88
3	Low-Level Twisted	89
3.1	Reactor Overview	89
3.1.1	Reactor Basics	89
3.1.2	Using the reactor object	89
3.2	UDP Networking	90
3.2.1	Overview	90
3.2.2	DatagramProtocol	90
3.2.3	Connected UDP	90
3.2.4	Multicast UDP	91
3.2.5	Acknowledgements	92
3.3	Using Processes	92
3.3.1	Overview	92
3.3.2	Running Another Process	93
3.3.3	Writing a ProcessProtocol	93
3.3.4	Things that can happen to your ProcessProtocol	94
3.3.5	Things you can do from your ProcessProtocol	95
3.3.6	Verbose Example	95

3.3.7	Doing it the Easy Way	96
3.3.8	Mapping File Descriptors	97
3.4	Deferred Reference	99
3.4.1	Callbacks	100
3.4.2	Errbacks	104
3.4.3	Handling either synchronous or asynchronous results	105
3.4.4	DeferredList	106
3.4.5	Class Overview	108
3.4.6	See also	108
3.5	Generating Deferreds	109
3.5.1	Class overview	109
3.5.2	What Deferreds don't do: make your code asynchronous	109
3.5.3	Advanced Processing Chain Control	110
3.5.4	Returning Deferreds from synchronous functions	110
3.5.5	Integrating blocking code with Twisted	111
3.5.6	Possible sources of error	112
3.6	Deferreds are beautiful! (A Tutorial)	112
3.6.1	Introduction	112
3.6.2	A simple example	113
3.6.3	Errbacks	114
3.6.4	addBoth: the deferred version of finally	121
3.6.5	addCallbacks: decision making based on previous success or failure	124
3.6.6	Hints, tips, common mistakes, and miscellany	129
3.6.7	Conclusion	133
3.7	Scheduling tasks for the future	133
3.8	Using Threads in Twisted	133
3.8.1	Running code in a thread-safe manner	133
3.8.2	Running code in threads	134
3.8.3	Utility Methods	134
3.8.4	Managing the Thread Pool	135
3.9	Choosing a Reactor and GUI Toolkit Integration	135
3.9.1	Overview	135
3.9.2	Reactor Functionality	136
3.9.3	General Purpose Reactors	136
3.9.4	Platform-Specific Reactors	136
3.9.5	GUI Integration Reactors	137
3.9.6	Non-Reactor GUI Integration	137
4	High-Level Twisted	139
4.1	The Basics	139
4.1.1	Application	139
4.1.2	twistd	139
4.1.3	tap2deb	139
4.1.4	tap2rpm	140
4.2	The Twisted Plugin System	140
4.2.1	Writing Extensible Programs	140
4.2.2	Extending an Existing Program	141
4.2.3	Alternate Plugin Packages	141
4.2.4	Plugin Caching	142
4.2.5	Further Reading	142
4.3	Writing a Twisted Application Plugin for twistd	142
4.3.1	Goals	143
4.3.2	A note on .tap files	143
4.3.3	Alternatives to TAP	143
4.3.4	Creating the plugin	143
4.3.5	Conclusion	144
4.4	Components: Interfaces and Adapters	144

4.4.1	Interfaces and Components in Twisted code	146
4.5	Cred: Pluggable Authentication	150
4.5.1	Goals	150
4.5.2	Cred objects	151
4.5.3	Responsibilities	153
4.6	Using the Twisted Application Framework	155
4.6.1	Introduction	155
4.6.2	Overview	155
4.6.3	Using application	156
5	Utilities	159
5.1	Using usage.Options	159
5.1.1	Introduction	159
5.1.2	Boolean Options	159
5.1.3	Parameters	160
5.1.4	Option Subcommands	161
5.1.5	Generic Code For Options	162
5.1.6	Parsing Arguments	162
5.1.7	Post Processing	163
5.2	Logging with twisted.python.log	163
5.2.1	Basic usage	163
5.2.2	Writing log observers	164
5.3	DirDBM: Directory-based Storage	164
5.3.1	dirdbm.DirDBM	164
5.3.2	dirdbm.Shelf	165
5.4	Using telnet to manipulate a twisted server	165
5.5	Writing tests for Twisted code	166
5.5.1	Trial basics	166
5.5.2	Twisted-specific quirks: reactor, Deferreds, callLater	166
6	Twisted RDBMS support	168
6.1	twisted.enterprise.adbapi: Twisted RDBMS support	168
6.1.1	Abstract	168
6.1.2	What you should already know	168
6.1.3	Quick Overview	168
6.1.4	How do I use adbapi?	169
6.1.5	Examples of various database adapters	170
6.1.6	And that's it!	170
6.2	Twisted Enterprise Row Objects	170
6.2.1	Class Definitions	170
6.2.2	Initialization	171
6.2.3	Creating Row Objects	171
6.2.4	Relationships Between Tables	172
6.2.5	Duplicate Row Objects	172
6.2.6	Updating Row Objects	172
6.2.7	Deleting Row Objects	172
7	Perspective Broker	173
7.1	Overview of Twisted Spread	173
7.1.1	Rationale	173
7.2	Introduction to Perspective Broker	173
7.2.1	Introduction	173
7.2.2	Object Roadmap	174
7.2.3	Things you can Call Remotely	174
7.2.4	Things you can Copy Remotely	175
7.3	Using Perspective Broker	176
7.3.1	Basic Example	176

7.3.2	Complete Example	178
7.3.3	Passing more references	180
7.3.4	References can come back to you	181
7.3.5	References to client-side objects	183
7.3.6	Raising Remote Exceptions	184
7.3.7	Try/Except blocks and Failure.trap	186
7.4	PB Copyable: Passing Complex Types	189
7.4.1	Overview	189
7.4.2	Motivation	189
7.4.3	Passing Objects	189
7.4.4	pb.Copyable	190
7.4.5	pb.Cacheable	196
7.5	Authentication with Perspective Broker	200
7.5.1	Overview	200
7.5.2	Compartmentalizing Services	200
7.5.3	Avatars and Perspectives	204
7.5.4	Perspective Examples	205
7.5.5	Using Avatars	208
8	Manual Pages	213
8.1	MANHOLE.1	213
8.1.1	NAME	213
8.1.2	SYNOPSIS	213
8.1.3	DESCRIPTION	213
8.1.4	AUTHOR	213
8.1.5	REPORTING BUGS	213
8.1.6	COPYRIGHT	213
8.2	MKTAP.1	214
8.2.1	NAME	214
8.2.2	SYNOPSIS	214
8.2.3	DESCRIPTION	214
8.2.4	portforward options	214
8.2.5	web options	214
8.2.6	toc options	214
8.2.7	mail options	214
8.2.8	telnet options	215
8.2.9	socks options	215
8.2.10	ftp options	215
8.2.11	manhole options	215
8.2.12	words options	215
8.2.13	AUTHOR	215
8.2.14	REPORTING BUGS	215
8.2.15	COPYRIGHT	215
8.2.16	SEE ALSO	215
8.3	TAP2DEB.1	216
8.3.1	NAME	216
8.3.2	SYNOPSIS	216
8.3.3	DESCRIPTION	216
8.3.4	AUTHOR	216
8.3.5	REPORTING BUGS	216
8.3.6	COPYRIGHT	216
8.3.7	SEE ALSO	216
8.4	TAP2RPM.1	217
8.4.1	NAME	217
8.4.2	SYNOPSIS	217
8.4.3	DESCRIPTION	217
8.4.4	AUTHOR	217

8.4.5	REPORTING BUGS	217
8.4.6	COPYRIGHT	217
8.4.7	SEE ALSO	217
8.5	TAPCONVERT.1	218
8.5.1	NAME	218
8.5.2	SYNOPSIS	218
8.5.3	DESCRIPTION	218
8.5.4	AUTHOR	218
8.5.5	REPORTING BUGS	218
8.5.6	COPYRIGHT	218
8.5.7	SEE ALSO	218
8.6	TRIAL.1	219
8.6.1	NAME	219
8.6.2	SYNOPSIS	219
8.6.3	DESCRIPTION	219
8.6.4	AUTHOR	219
8.6.5	REPORTING BUGS	219
8.6.6	COPYRIGHT	219
8.7	TWISTD.1	220
8.7.1	NAME	220
8.7.2	SYNOPSIS	220
8.7.3	DESCRIPTION	220
8.7.4	AUTHOR	220
8.7.5	REPORTING BUGS	220
8.7.6	COPYRIGHT	220
8.7.7	SEE ALSO	220
9	Appendix	221
9.1	The Twisted FAQ	221
9.1.1	General	221
9.1.2	Stability	221
9.1.3	Installation	222
9.1.4	Core Twisted	222
9.1.5	Requests and Contributing	224
9.1.6	Documentation	225
9.1.7	Communicating with us	225
9.2	Twisted Glossary	226
9.3	Banana Protocol Specifications	228
9.3.1	Introduction	228
9.3.2	Banana Encodings	228
9.3.3	Element Types	228
9.3.4	Profiles	229
9.3.5	Protocol Handshake and Behaviour	230

Chapter 1

Introduction

1.1 The Vision For Twisted

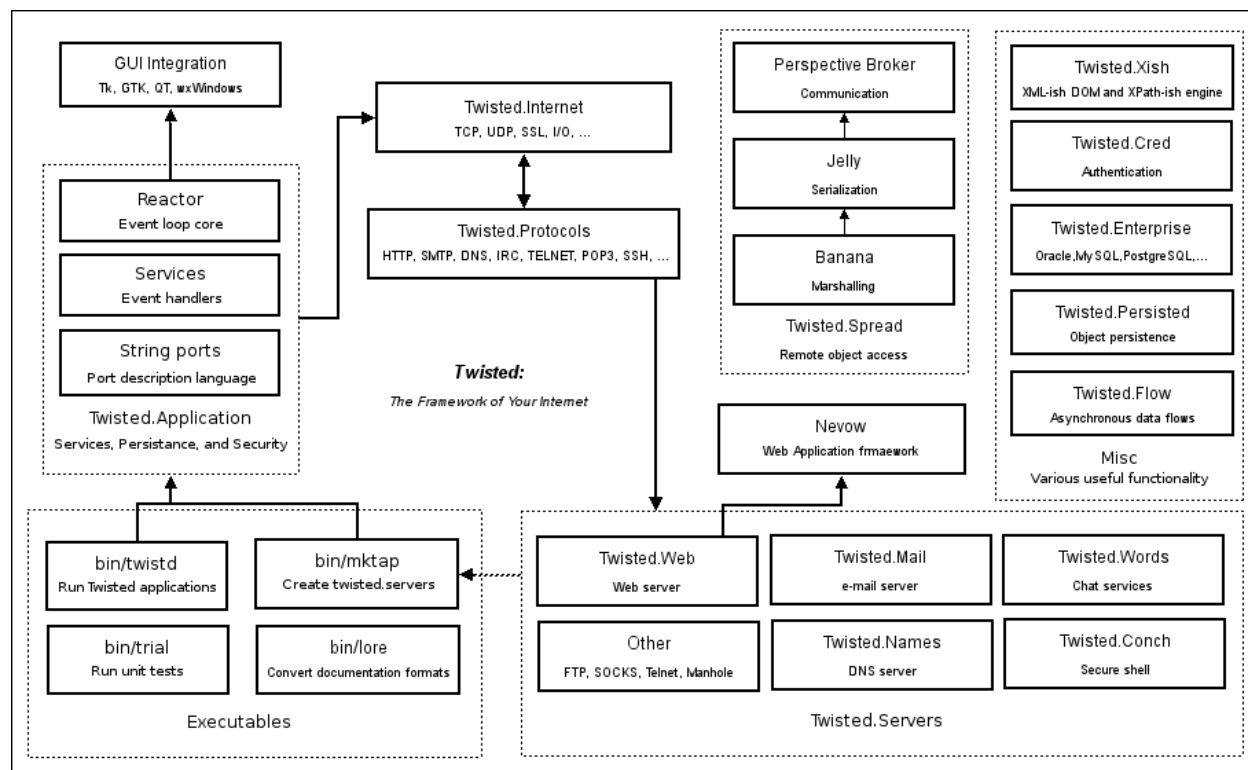
Many other documents in this repository are dedicated to defining what Twisted is. Here, I will attempt to explain not what Twisted is, but what it should be, once I've met my goals with it.

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

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

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

1.2 High-Level Overview of Twisted



1.3 Asynchronous Programming with Twisted

This document is a introduction to the asynchronous programming model, and to Twisted's Deferred abstraction, which symbolises a 'promised' result and which can pass an eventual result to handler functions.

This document is for readers new to Twisted who are familiar with the Python programming language and, at least conceptually, with core networking concepts such as servers, clients and sockets. This document will give you a high level overview of concurrent programming (interleaving several tasks) and of Twisted's concurrency model: *non-blocking code* or *asynchronous code*.

After discussing the concurrency model of which Deferreds are a part, it will introduce the methods of handling results when a function returns a Deferred object.

1.3.1 Introduction to concurrent programming

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

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

Waiting for answers

A fundamental feature of network programming is that of waiting for data. Imagine you have a function which sends an email summarising some information. This function needs to connect to a remote server, wait for the remote server to reply, check that the remote server can process the email, wait for the reply, send the email, wait for the confirmation, and then disconnect.

Any one of these steps may take a long period of time. Your program might use the simplest of all possible models, in which it actually sits and waits for data to be sent and received, but in this case it has some very obvious and basic limitations: it can't send many emails at once; and in fact it can't do anything else while it is sending an email.

Hence, all but the simplest network programs avoid this model. You can use one of several different models to allow your program to keep doing whatever tasks it has on hand while it is waiting for something to happen before a particular task can continue.

Not waiting on data

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

1. handle each connection in a separate operating system process, in which case the operating system will take care of letting other processes run while one is waiting;
2. handle each connection in a separate thread¹ in which the threading framework takes care of letting other threads run while one is waiting; or
3. use non-blocking system calls to handle all connections in one thread.

Non-blocking calls

The normal model when using the Twisted framework is the third model: non-blocking calls.

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.

In this model, the earlier email sending function would work something like this:

1. it calls a connection function to connect to the remote server;
2. the connection function returns immediately, with the implication that the notify the email sending library will be called when the connect has been made; and
3. once the connection is made, the connect mechanism notifies the email sending function that the connection is ready.

What advantage does the above sequence have over our original blocking sequence? The advantage is that while the email sending function can't do the next part of its job until the connection is open, the rest of the program can do other tasks, like begin the opening sequence for other email connections. Hence, the entire program is not waiting for the connection.

Callbacks

The typical asynchronous model for alerting an application that some data is ready for it is known as a *callback*. The application calls a function to request some data, and in this call, it also passes a callback function that should be called when the data is ready with the data as an argument. The callback function should therefore perform whatever tasks it was that the application needed that data for.

In synchronous programming, a function requests data, waits for the data, and then processes it. In asynchronous programming, a function requests the data, and lets the library call the callback function when the data is ready.

1.3.2 Deferreds

Twisted uses the `Deferred` object to manage the callback sequence. The client application attaches a series of functions to the deferred to be called in order when the results of the asynchronous request are available (this series of functions is known as a series of *callbacks*, or a *callback chain*), together with a series of functions to be called if there is an error in the asynchronous request (known as a series of *errbacks* or an *errback chain*). The asynchronous library code calls the first callback when the result is available, or the first errback when an error occurs, and the `Deferred` object then hands the results of each callback or errback function to the next function in the chain.

1.3.3 The Problem that Deferreds Solve

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

Deferreds are designed to enable Twisted programs to wait for data without hanging until that data arrives. They do this by giving a simple management interface for callbacks to libraries and applications. Libraries know that they

¹There 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.

always make their results available by calling `Deferred.callback` and errors by calling `Deferred.errback`. Applications set up result handlers by attaching callbacks and errbacks to deferreds in the order they want them called.

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

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

1.3.4 Deferreds - a signal that data is yet to come

In our email sending example above, a parent function calls a function to connect to the remote server. Asynchrony requires that this connection function return *without waiting for the result* so that the parent function can do other things. So how does the parent function or its controlling program know that the connection doesn't exist yet, and how does it use the connection once it does exist?

Twisted has an object that signals this situation. When the connection function returns, it signals that the operation is incomplete by returning a `twisted.internet.defer.Deferred` object.

The `Deferred` has two purposes. The first is that it says "I am a signal that the result of whatever you wanted me to do is still pending." The second is that you can ask the `Deferred` to run things when the data does arrive.

Callbacks

The way you tell a `Deferred` what to do with the data once it arrives is by adding a callback — asking the `Deferred` to call a function once the data arrives.

One Twisted library function that returns a `Deferred` is `twisted.web.client.getPage`. In this example, we call `getPage`, which returns a `Deferred`, and we attach a callback to handle the contents of the page once the data is available:

```
from twisted.web.client import getPage

from twisted.internet import reactor

def printContents(contents):
    '''
    This is the 'callback' function, added to the Deferred and called by
    it when the promised data is available
    '''

    print "The Deferred has called printContents with the following contents:"
    print contents

    # Stop the Twisted event handling system -- this is usually handled
    # in higher level ways
    reactor.stop()

# call getPage, which returns immediately with a Deferred, promising to
# pass the page contents onto our callbacks when the contents are available
deferred = getPage('http://twistedmatrix.com/')

# add a callback to the deferred -- request that it run printContents when
# the page content has been downloaded
deferred.addCallback(printContents)

# Begin the Twisted event handling system to manage the process -- again this
# isn't the usual way to do this
reactor.run()
```

A very common use of `Deferreds` is to attach two callbacks. The result of the first callback is passed to the second callback:

```

from twisted.web.client import getPage

from twisted.internet import reactor

def lowerCaseContents(contents):
    '''
    This is a 'callback' function, added to the Deferred and called by
    it when the promised data is available. It converts all the data to
    lower case
    '''

    return contents.lower()

def printContents(contents):
    '''
    This a 'callback' function, added to the Deferred after lowerCaseContents
    and called by it with the results of lowerCaseContents
    '''

    print contents
    reactor.stop()

deferred = getPage('http://twistedmatrix.com/')

# add two callbacks to the deferred -- request that it run lowerCaseContents
# when the page content has been downloaded, and then run printContents with
# the result of lowerCaseContents
deferred.addCallback(lowerCaseContents)
deferred.addCallback(printContents)

reactor.run()

```

Error handling: errbacks

Just as a asynchronous function returns before its result is available, it may also return before it is possible to detect errors: failed connections, erroneous data, protocol errors, and so on. Just as you can add callbacks to a Deferred which it calls when the data you are expecting is available, you can add error handlers ('errbacks') to a Deferred for it to call when an error occurs and it cannot obtain the data:

```

from twisted.web.client import getPage

from twisted.internet import reactor

def errorHandler(error):
    '''
    This is an 'errback' function, added to the Deferred which will call
    it in the event of an error
    '''

    # this isn't a very effective handling of the error, we just print it out:
    print "An error has occurred: <%s>" % str(error)
    # and then we stop the entire process:
    reactor.stop()

def printContents(contents):
    '''
    This a 'callback' function, added to the Deferred and called by it with
    the page content

```

```

'''

print contents
reactor.stop()

# We request a page which doesn't exist in order to demonstrate the
# error chain
deferred = getPage('http://twistedmatrix.com/does-not-exist')

# add the callback to the Deferred to handle the page content
deferred.addCallback(printContents)

# add the errback to the Deferred to handle any errors
deferred.addErrback(errorHandler)

reactor.run()
```

1.3.5 Conclusion

In this document, you have:

1. seen why non-trivial network programs need to have some form of concurrency;
2. learnt that the Twisted framework supports concurrency in the form of asynchronous calls;
3. learnt that the Twisted framework has Deferred objects that manage callback chains;
4. seen how the `getPage` function returns a Deferred object;
5. attached callbacks and errbacks to that Deferred; and
6. seen the Deferred's callback chain and errback chain fire.

See also

Since the Deferred abstraction is such a core part of programming with Twisted, there are several other detailed guides to it:

1. *Using Deferreds* (page 99), a more complete guide to using Deferreds, including Deferred chaining.
2. *Generating Deferreds* (page 109), a guide to creating Deferreds and firing their callback chains.

1.4 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.

Chapter 2

Tutorial

2.1 Writing Servers

2.1.1 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 140). This document is only relevant to TCP, SSL and Unix socket servers, there is a *separate document* (page 90) 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.

2.1.2 Protocols

As mentioned above, this, along with auxiliary classes and functions, is where most of the code is. A Twisted protocol handles data in an asynchronous manner. 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.transportloseConnection()
```

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.transportloseConnection()

    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 140) 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.transportloseConnection()

# 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, which returns it to using `lineReceived`.

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.

2.1.3 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.transportloseConnection()
```



```
def makeQOTDFactory(quote=None):
    factory = Factory()
    factory.protocol = QOTD
    factory.quote = quote or 'An apple a day keeps the doctor away'
    return factory
```

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

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

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

```
class LoggingProtocol(LineReceiver):

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

```
class LogfileFactory(Factory):

    protocol = LoggingProtocol

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

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

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

Putting it All Together

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

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

```
class QOTD(Protocol):

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

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 155).

2.2 Writing Clients

2.2.1 Overview

Twisted is a framework designed to be very flexible, and let you write powerful clients. The cost of this flexibility is a few layers in the way to writing your client. This document covers creating clients that can be used for TCP, SSL and Unix sockets, UDP is covered *in a different document* (page 90).

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

The persistent configuration is kept in a `Factory` class, which usually inherits from `twisted.internet.protocol.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.

2.2.2 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.transportloseConnection()
```

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

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

2.2.3 Simple, single-use clients

In many cases, the protocol only needs to connect to the server once, and the code just wants to get a connected instance of the protocol. In those cases `twisted.internet.protocol.ClientCreator` provides the appropriate API.

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

```

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

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

c = ClientCreator(reactor, Greeter)
c.connectTCP("localhost", 1234).addCallback(gotProtocol)

```

2.2.4 ClientFactory

We use `reactor.connect*` and a `ClientFactory`. The `ClientFactory` is in charge of creating the `Protocol`, and also receives events relating to the connection state. This allows it to do things like reconnect on the event of a connection error. Here is an example of a simple `ClientFactory` that uses the Echo protocol (above) and also prints what state the connection is in.

```

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

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

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

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

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

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

```

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

```

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

```

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

Reconnection

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

```

from twisted.internet.protocol import ClientFactory

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

```

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

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

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

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

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

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

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

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

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

2.2.5 A Higher-Level Example: ircLogBot

Overview of ircLogBot

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

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

# system imports
import time, sys

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

```

def log(self, message):
    """Write a message to the file."""
    timestamp = time.strftime("[%H:%M:%S]", time.localtime(time.time()))
    self.file.write('%s %s\n' % (timestamp, message))
    self.file.flush()

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

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

    nickname = "twistedbot"

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

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

    # callbacks for events

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

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

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

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

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

    def action(self, user, channel, msg):
        """This will get called when the bot sees someone do an action."""

```

```

        user = user.split('!', 1)[0]
        self.logger.log("* %s %s" % (user, msg))

# irc callbacks

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

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

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

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

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

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

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

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

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

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

    # run bot
    reactor.run()

```

Source listing — *ircLogBot.py*

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

Persistent Data in the Factory

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

```

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

class LogBot(irc.IRCClient):

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

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

class LogBotFactory(protocol.ClientFactory):

    protocol = LogBot

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

```

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

2.3 Setting up the TwistedQuotes application

2.3.1 Goal

This document describes how to set up the TwistedQuotes application used in a number of other documents, such as *designing Twisted applications* (page 23).

2.3.2 Setting up the TwistedQuotes project directory

In order to run the Twisted Quotes example, you will need to do the following:

1. Make a TwistedQuotes directory on your system
2. Place the following files in the TwistedQuotes directory:
 - `__init__.py` (page ??) (this file marks it as a package, see this section¹ of the Python tutorial for more on packages);
 - `quoters.py` (page ??);
 - `quoteproto.py` (page ??); and
 - `plugins.tml` (page ??).
3. Add the TwistedQuotes directory's *parent* to your Python path. For example, if the TwistedQuotes directory's path is `/tmp/TwistedQuotes` add `/tmp` to your Python path. On UNIX this would be `export PYTHONPATH=/my/stuff:$PYTHONPATH`, on Microsoft Windows change the `PYTHONPATH` variable through the Systems Properties dialog to add `/my/stuff`; at the beginning.
4. Test your package by trying to import it in the Python interpreter:

```

Python 2.1.3 (#1, Apr 20 2002, 22:45:31)
[GCC 2.95.4 20011002 (Debian prerelease)] on linux2

```

¹<http://www.python.org/doc/current/tut/node8.html#SECTION00840000000000000000>

```
Type "copyright", "credits" or "license" for more information.
>>> import TwistedQuotes
>>> # No traceback means you're fine.
```

2.4 Designing Twisted Applications

2.4.1 Goals

This document describes how a good Twisted application is structured. It should be useful for beginning Twisted developers who want to structure their code in a clean, maintainable way that reflects current best practices.

Readers will want to be familiar with *asynchronous programming using Deferreds* (page 8) and with writing *servers* (page 13) and *clients* (page 17) using Twisted.

2.4.2 Example of a modular design: TwistedQuotes

TwistedQuotes is a very simple plugin which is a great demonstration of Twisted's power. It will export a small kernel of functionality – Quote of the Day – which can be accessed through every interface that Twisted supports: web pages, e-mail, instant messaging, a specific Quote of the Day protocol, and more.

Set up the project directory

See the description of *setting up the TwistedQuotes example* (page 22).

A Look at the Heart of the Application

```
from zope.interface import Interface, implements
```

```
from random import choice
```

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

```
    def getQuote():
        """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 99).

Your Twisted applications should follow this style as much as possible. Have (at least) one module which implements your specific functionality, independent of any user-interface code.

Next, we're going to need to associate this abstract logic with some way of displaying it to the user. We'll do this by writing a Twisted server protocol, which will respond to the clients that connect to it by sending a quote to the client and then closing the connection. Note: don't get too focused on the details of this – different ways to interface with the user are 90% of what Twisted does, and there are lots of documents describing the different ways to do it.

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

class QOTD(Protocol):

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

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 `mktaap` plug-in below.

Note: you can read more specifics of Protocol and Factory in the *Writing Servers* (page 13) HOWTO.

Once we have an abstraction – a Quoter – and we have a mechanism to connect it to the network – the QOTD protocol – the next thing to do is to put the last link in the chain of functionality between abstraction and user. This last link will allow a user to choose a Quoter and configure the protocol. Writing this configuration is covered in the *Application HOWTO* (page 155).

2.5 Twisted from Scratch, or The Evolution of Finger

2.5.1 Introduction

Twisted is a big system. People are often daunted when they approach it. It's hard to know where to start looking.

This guide builds a full-fledged Twisted application from the ground up, using most of the important bits of the framework. There is a lot of code, but don't be afraid.

The application we are looking at is a "finger" service, along the lines of the familiar service traditionally provided by UNIX servers. We will extend this service slightly beyond the standard, in order to demonstrate some of Twisted's higher-level features.

2.5.2 Contents

This tutorial is split into eleven parts:

1. *The Evolution of Finger: building a simple finger service* (this page)
2. *The Evolution of Finger: adding features to the finger service* (page 30)
3. *The Evolution of Finger: cleaning up the finger code* (page 37)
4. *The Evolution of Finger: moving to a component based architecture* (page 40)
5. *The Evolution of Finger: pluggable backends* (page 50)
6. *The Evolution of Finger: a web frontend* (page 61)
7. *The Evolution of Finger: Twisted client support using Perspective Broker* (page 65)
8. *The Evolution of Finger: using a single factory for multiple protocols* (page 71)
9. *The Evolution of Finger: a Twisted finger client* (page 77)
10. *The Evolution of Finger: making a finger library* (page 79)
11. *The Evolution of Finger: configuration and packaging of the finger service* (page 81)

2.6 The Evolution of Finger: building a simple finger service

2.6.1 Introduction

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

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

2.6.2 Refuse Connections

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

Source listing — *finger01.py*

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

The Reactor

You don't call Twisted, Twisted calls you. The `reactor` is Twisted's main event loop. There is exactly one reactor in any running Twisted application. Once started it loops over and over again, responding to network events, and making scheduled calls to code.

2.6.3 Do Nothing

```
from twisted.internet import protocol, reactor
class FingerProtocol(protocol.Protocol):
    pass
class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol
reactor.listenTCP(1079, FingerFactory())
reactor.run()
```

Source listing — *finger02.py*

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

2.6.4 Drop Connections

```
from twisted.internet import protocol, reactor
class FingerProtocol(protocol.Protocol):
    def connectionMade(self):
        self.transport.loseConnection()
class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol
reactor.listenTCP(1079, FingerFactory())
reactor.run()
```

Source listing — *finger03.py*

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

2.6.5 Read Username, Drop Connections

```
from twisted.internet import protocol, reactor
from twisted.protocols import basic
class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.transport.loseConnection()
class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol
reactor.listenTCP(1079, FingerFactory())
reactor.run()
```

Source listing — *finger04.py*

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

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

2.6.6 Read Username, Output Error, Drop Connections

```
from twisted.internet import protocol, reactor
from twisted.protocols import basic
class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.transport.write("No such user\r\n")
        self.transport.loseConnection()
class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol
reactor.listenTCP(1079, FingerFactory())
reactor.run()
```

Source listing — *finger05.py*

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

2.6.7 Output From Empty Factory

```
# Read username, output from empty factory, drop connections
from twisted.internet import protocol, reactor
from twisted.protocols import basic
class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.transport.write(self.factory.getUser(user)+"\r\n")
        self.transport.loseConnection()
class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol
    def getUser(self, user): return "No such user"
reactor.listenTCP(1079, FingerFactory())
reactor.run()
```

Source listing — *finger06.py*

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

2.6.8 Output from Non-empty Factory

```
# Read username, output from non-empty factory, drop connections
from twisted.internet import protocol, reactor
from twisted.protocols import basic
class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.transport.write(self.factory.getUser(user)+"\r\n")
        self.transport.loseConnection()
class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol
    def __init__(self, **kwargs): self.users = kwargs
    def getUser(self, user):
        return self.users.get(user, "No such user")
reactor.listenTCP(1079, FingerFactory(moshez='Happy and well'))
reactor.run()
```

Source listing — *finger07.py*

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

2.6.9 Use Deferreds

```
# Read username, output from non-empty factory, drop connections
# Use deferreds, to minimize synchronicity assumptions
from twisted.internet import protocol, reactor, defer
from twisted.protocols import basic
class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.factory.getUser(user
        ).addErrback(lambda _: "Internal error in server"
        ).addCallback(lambda m:
            (self.transport.write(m+"\r\n"),
             self.transportloseConnection()))
class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol
    def __init__(self, **kwargs): self.users = kwargs
    def getUser(self, user):
        return defer.succeed(self.users.get(user, "No such user"))
reactor.listenTCP(1079, FingerFactory(moshez='Happy and well'))
reactor.run()
```

Source listing — *finger08.py*

But, here we tweak it just for the hell of it. Yes, while the previous version worked, it did assume the result of `getUser` is always immediately available. But what if instead of an in memory database, we would have to fetch result from a remote Oracle? Or from the web? Or, or...

2.6.10 Run 'finger' Locally

```
# Read username, output from factory interfacing to OS, drop connections
from twisted.internet import protocol, reactor, defer, utils
from twisted.protocols import basic
class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.factory.getUser(user
        ).addErrback(lambda _: "Internal error in server"
        ).addCallback(lambda m:
            (self.transport.write(m+"\r\n"),
             self.transportloseConnection()))
class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol
    def getUser(self, user):
        return utils.getProcessOutput("finger", [user])
reactor.listenTCP(1079, FingerFactory())
reactor.run()
```

Source listing — *finger09.py*

...from running a local command? Yes, this version runs finger locally with whatever arguments it is given, and returns the standard output. This is probably insecure, so you probably don't want a real server to do this without a lot more validation of the user input. This will do exactly what the standard version of the finger server does.

2.6.11 Read Status from the Web

The web. That invention which has infiltrated homes around the world finally gets through to our invention. Here we use the built-in Twisted web client, which also returns a deferred. Finally, we manage to have examples of three different database back-ends, which do not change the protocol class. In fact, we will not have to change the protocol again until the end of this tutorial: we have achieved, here, one truly usable class.

```
# Read username, output from factory interfacing to web, drop connections
from twisted.internet import protocol, reactor, defer, utils
from twisted.protocols import basic
from twisted.web import client
class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.factory.getUser(user
        ).addErrback(lambda _: "Internal error in server"
        ).addCallback(lambda m:
                        (self.transport.write(m+"\r\n"),
                         self.transportloseConnection()))
class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol
    def __init__(self, prefix): self.prefix=prefix
    def getUser(self, user):
        return client.getPage(self.prefix+user)
reactor.listenTCP(1079, FingerFactory(prefix='http://livejournal.com/~'))
reactor.run()
```

Source listing — *finger10.py*

2.6.12 Use Application

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

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

```
# Read username, output from non-empty factory, drop connections
# Use deferreds, to minimize synchronicity assumptions
# Write application. Save in 'finger.tpy'
from twisted.application import internet, service
from twisted.internet import protocol, reactor, defer
from twisted.protocols import basic
class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.factory.getUser(user
        ).addErrback(lambda _: "Internal error in server"
        ).addCallback(lambda m:
                        (self.transport.write(m+"\r\n"),
                         self.transportloseConnection()))
class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol
```

```

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

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

```

Source listing — *finger11.py*

2.6.13 twistd

This is how to run “Twisted Applications”— files which define an ‘application’. twistd (TWISTed Daemonizer) does everything a daemon can be expected to — shuts down stdin/stdout/stderr, disconnects from the terminal and can even change runtime directory, or even the root filesystems. In short, it does everything so the Twisted application developer can concentrate on writing his networking code.

```

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

```

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

2.7.1 Introduction

This is the second part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 25).

In this section of the tutorial, our finger server will continue to sprout features: the ability for users to set finger announces, and using our finger service to send those announcements on the web, on IRC and over XML-RPC.

2.7.2 Setting Message By Local Users

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

```

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

# But let's try and fix setting away messages, shall we?
from twisted.application import internet, service
from twisted.internet import protocol, reactor, defer
from twisted.protocols import basic
class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.factory.getUser(user
        ).addErrback(lambda _: "Internal error in server"
        ).addCallback(lambda m:
            (self.transport.write(m+"\r\n"),
             self.transportloseConnection()))

```

```

class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol
    def __init__(self, **kwargs): self.users = kwargs
    def getUser(self, user):
        return defer.succeed(self.users.get(user, "No such user"))

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

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

ff = FingerFactory(moshez='Happy and well')
fsf = FingerSetterFactory(ff)

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

```

Source listing — *finger12.py*

2.7.3 Use Services to Make Dependencies Sane

The previous version had the setter poke at the innards of the finger factory. It's usually not a good idea: this version makes both factories symmetric by making them both look at a single object. Services are useful for when an object is needed which is not related to a specific network server. Here, we moved all responsibility for manufacturing factories into the service. Note that we stopped subclassing: the service simply puts useful methods and attributes inside the factories. We are getting better at protocol design: none of our protocol classes had to be changed, and neither will have to change until the end of the tutorial.

```

# Fix asymmetry
from twisted.application import internet, service
from twisted.internet import protocol, reactor, defer
from twisted.protocols import basic
class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.factory.getUser(user
            ).addErrback(lambda _: "Internal error in server"
            ).addCallback(lambda m:
                (self.transport.write(m+"\r\n"),
                 self.transportloseConnection()))

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

class FingerService(service.Service):

```



```

def __init__(self, *args, **kwargs):
    self.parent.__init__(self, *args)
    self.users = kwargs
def getUser(self, user):
    return defer.succeed(self.users.get(user, "No such user"))
def getFingerFactory(self):
    f = protocol.ServerFactory()
    f.protocol, f.getUser = FingerProtocol, self.getUser
    return f
def getFingerSetterFactory(self):
    f = protocol.ServerFactory()
    f.protocol, f.setUser = FingerSetterProtocol, self.users.__setitem__
    return f

application = service.Application('finger', uid=1, gid=1)
f = FingerService('finger', moshez='Happy and well')
serviceCollection = service.IServiceCollection(application)
internet.TCPServer(79, f.getFingerFactory()
                    ).setServiceParent(serviceCollection)
internet.TCPServer(1079, f.getFingerSetterFactory()
                    ).setServiceParent(serviceCollection)

```

Source listing — *finger13.py*

2.7.4 Read Status File

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

```

moshez: happy and well
shawn: alive

```

sample /etc/users file — *etc.users*

```

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

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.factory.getUser(user)
        .addErrback(lambda _: "Internal error in server")
        .addCallback(lambda m:
                      (self.transport.write(m+"\r\n"),
                       self.transportloseConnection()))

class FingerService(service.Service):
    def __init__(self, filename):
        self.users = {}
        self.filename = filename
    def _read(self):
        for line in file(self.filename):
            user, status = line.split(':', 1)
            user = user.strip()
            status = status.strip()

```

```

        self.users[user] = status
    self.call = reactor.callLater(30, self._read)
def startService(self):
    self._read()
    service.Service.startService(self)
def stopService(self):
    service.Service.stopService(self)
    self.call.cancel()
def getUser(self, user):
    return defer.succeed(self.users.get(user, "No such user"))
def getFingerFactory(self):
    f = protocol.ServerFactory()
    f.protocol, f.getUser = FingerProtocol, self.getUser
    return f

application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
finger = internet.TCPServer(79, f.getFingerFactory())

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

```

Source listing — *finger14.py*

2.7.5 Announce on Web, Too

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

```

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

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.factory.getUser(user)
        ).addErrback(lambda _: "Internal error in server")
        ).addCallback(lambda m:
            (self.transport.write(m+"\r\n"),
             self.transportloseConnection()))

class MotdResource(resource.Resource):

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

    # we treat the path as the username
    def getChild(self, username, request):
        motd = self.users.get(username)
        username = cgi.escape(username)
        if motd is not None:

```

```

        motd = cgi.escape(motd)
        text = '<h1>%s</h1><p>%s</p>' % (username, motd)
    else:
        text = '<h1>%s</h1><p>No such user</p>' % username
    return static.Data(text, 'text/html')

class FingerService(service.Service):
    def __init__(self, filename):
        self.filename = filename
        self._read()
    def _read(self):
        self.users = {}
        for line in file(self.filename):
            user, status = line.split(':', 1)
            user = user.strip()
            status = status.strip()
            self.users[user] = status
        self.call = reactor.callLater(30, self._read)
    def getUser(self, user):
        return defer.succeed(self.users.get(user, "No such user"))
    def getFingerFactory(self):
        f = protocol.ServerFactory()
        f.protocol, f.getUser = FingerProtocol, self.getUser
        f.startService = self.startService
        return f

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

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

```

Source listing — *finger15.py*

2.7.6 Announce on IRC, Too

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

```

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

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

```

```

        self.factory.getUser(user
    ).addErrback(lambda _: "Internal error in server"
    ).addCallback(lambda m:
        (self.transport.write(m+"\r\n"),
         self.transport.looseConnection()))
class FingerSetterProtocol(basic.LineReceiver):
    def connectionMade(self): self.lines = []
    def lineReceived(self, line): self.lines.append(line)
    def connectionLost(self, reason): self.factory.setUser(*self.lines[:2])
class IRCReplyBot(irc.IRCClient):
    def connectionMade(self):
        self.nickname = self.factory.nickname
        irc.IRCClient.connectionMade(self)
    def privmsg(self, user, channel, msg):
        user = user.split('!')[0]
        if self.nickname.lower() == channel.lower():
            self.factory.getUser(msg
            ).addErrback(lambda _: "Internal error in server"
            ).addCallback(lambda m: irc.IRCClient.msg(self, user, msg+' ': '+m))

class FingerService(service.Service):
    def __init__(self, filename):
        self.filename = filename
        self._read()
    def _read(self):
        self.users = {}
        for line in file(self.filename):
            user, status = line.split(':', 1)
            user = user.strip()
            status = status.strip()
            self.users[user] = status
        self.call = reactor.callLater(30, self._read)
    def getUser(self, user):
        return defer.succeed(self.users.get(user, "No such user"))
    def getFingerFactory(self):
        f = protocol.ServerFactory()
        f.protocol, f.getUser = FingerProtocol, self.getUser
        return f
    def getResource(self):
        r = resource.Resource()
        r.getChild = (lambda path, request:
            static.Data('<h1>%s</h1><p>%s</p>' %
            tuple(map(cgi.escape,
            [path, self.users.get(path,
            "No such user <p/> usage: site/user"]))),
            'text/html'))
        return r

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

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

```

```

        ).setServiceParent(serviceCollection)
internet.TCPServer(8000, server.Site(f.getResource()))
        ).setServiceParent(serviceCollection)
internet.TCPClient('irc.freenode.org', 6667, f.getIRCBot('fingerbot'))
        ).setServiceParent(serviceCollection)

```

Source listing — *finger16.py*

2.7.7 Add XML-RPC Support

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

```

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

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.factory.getUser(user
            ).addErrback(lambda _: "Internal error in server"
            ).addCallback(lambda m:
                (self.transport.write(m+"\r\n"),
                 self.transportloseConnection()))

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

class IRCReplyBot(irc.IRCClient):
    def connectionMade(self):
        self.nickname = self.factory.nickname
        irc.IRCClient.connectionMade(self)
    def privmsg(self, user, channel, msg):
        user = user.split('!')[0]
        if self.nickname.lower() == channel.lower():
            self.factory.getUser(msg
                ).addErrback(lambda _: "Internal error in server"
                ).addCallback(lambda m: irc.IRCClient.msg(self, user, msg+' '+m))

class FingerService(service.Service):
    def __init__(self, filename):
        self.filename = filename
        self._read()
    def _read(self):
        self.users = {}
        for line in file(self.filename):
            user, status = line.split(':', 1)
            user = user.strip()
            status = status.strip()
            self.users[user] = status
        self.call = reactor.callLater(30, self._read)
    def getUser(self, user):

```

```

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

application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
serviceCollection = service.IServiceCollection(application)
internet.TCPServer(79, f.getFingerFactory()
    ).setServiceParent(serviceCollection)
internet.TCPServer(8000, server.Site(f.getResource()
    ).setServiceParent(serviceCollection)
internet.TCPClient('irc.freenode.org', 6667, f.getIRCBot('fingerbot')
    ).setServiceParent(serviceCollection)

```

Source listing — *finger17.py*

A simple client to test the XMLRPC finger:

```

# testing xmlrpc finger

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

```

Source listing — *fingerXRclient.py*

2.8 The Evolution of Finger: cleaning up the finger code

2.8.1 Introduction

This is the third part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 25).

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

2.8.2 Write Readable Code

The last version of the application had a lot of hacks. We avoided sub-classing, didn't support things like user listings over the web, and removed all blank lines – all in the interest of code which is shorter. Here we take a step back, subclass what is more naturally a subclass, make things which should take multiple lines take them, etc. This shows a

much better style of developing Twisted applications, though the hacks in the previous stages are sometimes used in throw-away prototypes.

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

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

class FingerProtocol(basic.LineReceiver):

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

class FingerSetterProtocol(basic.LineReceiver):

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

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

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

class IRCReplyBot(irc.IRCClient):

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

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

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

    def render_GET(self, request):
```

```

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

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

class UserStatus(resource.Resource):

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

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

class UserStatusXR(xmlrpc.XMLRPC):

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

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

class FingerService(service.Service):

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

    def _read(self):
        self.users = {}
        for line in file(self.filename):
            user, status = line.split(':', 1)
            user = user.strip()
            status = status.strip()
            self.users[user] = status
        self.call = reactor.callLater(30, self._read)

```



```

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

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

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

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

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

application = service.Application('finger', uid=1, gid=1)
f = FingerService('/etc/users')
serviceCollection = service.IServiceCollection(application)
internet.TCPServer(79, f.getFingerFactory())
    .setServiceParent(serviceCollection)
internet.TCPServer(8000, server.Site(f.getResource()))
    .setServiceParent(serviceCollection)
internet.TCPClient('irc.freenode.org', 6667, f.getIRCBot('fingerbot'))
    .setServiceParent(serviceCollection)

```

Source listing — *finger18.py*

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

2.9.1 Introduction

This is the fourth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 25).

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

2.9.2 Write Maintainable Code

In the last version, the service class was three times longer than any other class, and was hard to understand. This was because it turned out to have multiple responsibilities. It had to know how to access user information, by rereading the file every half minute, but also how to display itself in a myriad of protocols. Here, we used the component-based architecture that Twisted provides to achieve a separation of concerns. All the service is responsible for, now, is supporting `getUser/getUsers`. It declares its support via a call to `zope.interface.implements`. Then, adapters are used to make this service look like an appropriate class for various things: for supplying a finger factory to `TCPServer`, for

supplying a resource to site's constructor, and to provide an IRC client factory for TCPClient. All the adapters use are the methods in FingerService they are declared to use: getUser/getUsers. We could, of course, skip the interfaces and let the configuration code use things like FingerFactoryFromService(f) directly. However, using interfaces provides the same flexibility inheritance gives: future subclasses can override the adapters.

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

class IFingerService(Interface):

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

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

class IFingerSetterService(Interface):

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

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

class FingerProtocol(basic.LineReceiver):

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

class IFingerFactory(Interface):

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

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

class FingerFactoryFromService(protocol.ServerFactory):

    implements(IFingerFactory)

    protocol = FingerProtocol

    def __init__(self, service):
```

```

        self.service = service

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

components.registerAdapter(FingerFactoryFromService,
                           IFingerService,
                           IFingerFactory)

class FingerSetterProtocol(basic.LineReceiver):

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

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

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

class IFingerSetterFactory(Interface):

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

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

class FingerSetterFactoryFromService(protocol.ServerFactory):

    implements(IFingerSetterFactory)

    protocol = FingerSetterProtocol

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

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

components.registerAdapter(FingerSetterFactoryFromService,
                           IFingerSetterService,
                           IFingerSetterFactory)

class IRCReplyBot(irc.IRCClient):

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

    def privmsg(self, user, channel, msg):
        user = user.split('!')[0]
        if self.nickname.lower() == channel.lower():

```

```

        d = self.factory.getUser(msg)
        d.addErrback(catchError)
        d.addCallback(lambda m: "Status of %s: %s" % (msg, m))
        d.addCallback(lambda m: self.msg(user, m))

class IIRCClientFactory(Interface):

    """
    @ivar nickname
    """

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

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

class IRCCClientFactoryFromService(protocol.ClientFactory):

    implements(IIRCCClientFactory)

    protocol = IRCReplyBot
    nickname = None

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

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

components.registerAdapter(IRCCClientFactoryFromService,
                           IFingerService,
                           IIRCCClientFactory)

class UserStatusTree(resource.Resource):

    implements(resource.IResource)

    def __init__(self, service):
        resource.Resource.__init__(self)
        self.service = service
        self.putChild('RPC2', UserStatusXR(self.service))

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

    def getChild(self, path, request):

```

```

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

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

class UserStatus(resource.Resource):

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

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

class UserStatusXR(xmlrpc.XMLRPC):

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

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

class FingerService(service.Service):

    implements(IFingerService)

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

    def _read(self):
        self.users = {}
        for line in file(self.filename):
            user, status = line.split(':', 1)
            user = user.strip()
            status = status.strip()
            self.users[user] = status
        self.call = reactor.callLater(30, self._read)

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

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

```

```

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

```

Source listing — *finger19.py*

2.9.3 Advantages of Latest Version

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

```

class IFingerSetterService(Interface):

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

# Advantages of latest version

class MemoryFingerService(service.Service):

    implements([IFingerService, IFingerSetterService])

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

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

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

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

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

```

Source listing — *finger19a_changes.py*

Full source code here:

```

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

class IFingerService(Interface):

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

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

class IFingerSetterService(Interface):

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

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

class FingerProtocol(basic.LineReceiver):

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

class IFingerFactory(Interface):

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

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

class FingerFactoryFromService(protocol.ServerFactory):

    implements(IFingerFactory)

    protocol = FingerProtocol

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

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

```

```

components.registerAdapter(FingerFactoryFromService,
                           IFingerService,
                           IFingerFactory)

class FingerSetterProtocol(basic.LineReceiver):

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

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

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

class IFingerSetterFactory(Interface):

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

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

class FingerSetterFactoryFromService(protocol.ServerFactory):

    implements(IFingerSetterFactory)

    protocol = FingerSetterProtocol

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

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

components.registerAdapter(FingerSetterFactoryFromService,
                           IFingerSetterService,
                           IFingerSetterFactory)

class IRCReplyBot(irc.IRCClient):

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

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

```



```

class IIRCClientFactory(Interface):

    """
    @ivar nickname
    """

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

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

class IRCCClientFactoryFromService(protocol.ClientFactory):

    implements(IIRCCClientFactory)

    protocol = IRCReplyBot
    nickname = None

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

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

components.registerAdapter(IRCCClientFactoryFromService,
                           IFingerService,
                           IIRCCClientFactory)

class UserStatusTree(resource.Resource):

    implements(resource.IResource)

    def __init__(self, service):
        resource.Resource.__init__(self)
        self.service = service
        self.putChild('RPC2', UserStatusXR(self.service))

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

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

```

```

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

class UserStatus(resource.Resource):

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

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

class UserStatusXR(xmlrpc.XMLRPC):

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

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

class MemoryFingerService(service.Service):

    implements([IFingerService, IFingerSetterService])

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

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

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

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

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

```

```

        ).setServiceParent(serviceCollection)
internet.TCPServer(1079, IFingerSetterFactory(f), interface='127.0.0.1'
        ).setServiceParent(serviceCollection)

```

Source listing — *finger19a.py*

2.9.4 Aspect-Oriented Programming

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

2.10 The Evolution of Finger: pluggable backends

2.10.1 Introduction

This is the fifth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 25).

In this part we will add new several new backends to our finger service using the component-based architecture developed in *The Evolution of Finger: moving to a component based architecture* (page 40). This will show just how convenient it is to implement new back-ends when we move to a component based architecture. Note that here we also use an interface we previously wrote, `FingerSetterFactory`, by supporting one single method. We manage to preserve the service's ignorance of the network.

2.10.2 Another Back-end

```

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

# Another back-end

class LocalFingerService(service.Service):

    implements(IFingerService)

    def getUser(self, user):
        # need a local finger daemon running for this to work
        return utils.getProcessOutput("finger", [user])

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

f = LocalFingerService()

```

Source listing — *finger19b_changes.py*

Full source code here:

```

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

```

```

import pwd

class IFingerService(Interface):

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

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

class IFingerSetterService(Interface):

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

class IFingerSetterService(Interface):

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

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

class FingerProtocol(basic.LineReceiver):

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

class IFingerFactory(Interface):

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

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

class FingerFactoryFromService(protocol.ServerFactory):

    implements(IFingerFactory)

    protocol = FingerProtocol

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

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

components.registerAdapter(FingerFactoryFromService,
                           IFingerService,

```

```

        IFingerFactory)

class FingerSetterProtocol(basic.LineReceiver):

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

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

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

class IFingerSetterFactory(Interface):

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

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

class FingerSetterFactoryFromService(protocol.ServerFactory):

    implements(IFingerSetterFactory)

    protocol = FingerSetterProtocol

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

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

components.registerAdapter(FingerSetterFactoryFromService,
                           IFingerSetterService,
                           IFingerSetterFactory)

class IRCReplyBot(irc.IRCClient):

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

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

class IIRCClientFactory(Interface):

```

```

"""
@ivar nickname
"""

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

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

class IRCCliientFactoryFromService(protocol.ClientFactory):

    implements(IIRCCliientFactory)

    protocol = IRCReplyBot
    nickname = None

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

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

components.registerAdapter(IRCCliientFactoryFromService,
                           IFingerService,
                           IIRCCliientFactory)

class UserStatusTree(resource.Resource):

    implements(resource.IResource)

    def __init__(self, service):
        resource.Resource.__init__(self)
        self.service = service
        self.putChild('RPC2', UserStatusXR(self.service))

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

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

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

```

```

class UserStatus(resource.Resource):

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

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

class UserStatusXR(xmlrpc.XMLRPC):

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

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

class FingerService(service.Service):

    implements(IFingerService)

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

    def _read(self):
        self.users = {}
        for line in file(self.filename):
            user, status = line.split(':', 1)
            user = user.strip()
            status = status.strip()
            self.users[user] = status
        self.call = reactor.callLater(30, self._read)

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

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

# Another back-end

class LocalFingerService(service.Service):

    implements(IFingerService)

```

```

def getUser(self, user):
    # need a local finger daemon running for this to work
    return utils.getProcessOutput("finger", [user])

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

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

```

Source listing — *finger19b.py*

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

2.10.3 Yet Another Back-end: Doing the Standard Thing

```

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

```

```

# Yet another back-end

```

```

class LocalFingerService(service.Service):

    implements(IFingerService)

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

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

```



```
f = LocalFingerService()
```

Source listing — *finger19c_changes.py*

Full source code here:

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

class IFingerService(Interface):

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

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

class IFingerSetterService(Interface):

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

class IFingerSetterService(Interface):

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

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

class FingerProtocol(basic.LineReceiver):

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

class IFingerFactory(Interface):

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

    def buildProtocol(addr):
```

```

        """Return a protocol returning a string"""

class FingerFactoryFromService(protocol.ServerFactory):

    implements(IFingerFactory)

    protocol = FingerProtocol

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

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

components.registerAdapter(FingerFactoryFromService,
                           IFingerService,
                           IFingerFactory)

class FingerSetterProtocol(basic.LineReceiver):

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

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

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

class IFingerSetterFactory(Interface):

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

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

class FingerSetterFactoryFromService(protocol.ServerFactory):

    implements(IFingerSetterFactory)

    protocol = FingerSetterProtocol

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

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

components.registerAdapter(FingerSetterFactoryFromService,
                           IFingerSetterService,
                           IFingerSetterFactory)

```

```

class IRCReplyBot(irc.IRCClient):

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

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

class IIRCClientFactory(Interface):

    """
    @ivar nickname
    """

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

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

class IRCClientFactoryFromService(protocol.ClientFactory):

    implements(IIRCClientFactory)

    protocol = IRCReplyBot
    nickname = None

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

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

components.registerAdapter(IRCClientFactoryFromService,
                           IFingerService,
                           IIRCClientFactory)

class UserStatusTree(resource.Resource):

    implements(resource.IResource)

    def __init__(self, service):
        resource.Resource.__init__(self)
        self.service = service
        self.putChild('RPC2', UserStatusXR(self.service))

    def render_GET(self, request):
        d = self.service.getUsers()

```

```

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

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

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

class UserStatus(resource.Resource):

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

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

class UserStatusXR(xmlrpc.XMLRPC):

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

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

class FingerService(service.Service):

    implements(IFingerService)

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

    def _read(self):
        self.users = {}
        for line in file(self.filename):
            user, status = line.split(':', 1)

```

```

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

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

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

# Yet another back-end

class LocalFingerService(service.Service):

    implements(IFingerService)

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

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

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

```

Source listing — *finger19c.py*

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

2.11 The Evolution of Finger: a web frontend

2.11.1 Introduction

This is the sixth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 25).

In this part, we demonstrate adding a web frontend using simple `twisted.web.resource.Resource` objects: `UserStatusTree`, which will produce a listing of all users at the base URL (`/`) of our site; `UserStatus`, which gives the status of each user at the location `/username`; and `UserStatusXR`, which exposes an XMLRPC interface to `getUser` and `getUsers` functions at the URL `/RPC2`.

In this example we construct HTML segments manually. If the web interface was less trivial, we would want to use more sophisticated web templating and design our system so that HTML rendering and logic were clearly separated.

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

class IFingerService(Interface):

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

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

class IFingerSetterService(Interface):

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

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

class FingerProtocol(basic.LineReceiver):

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

class IFingerFactory(Interface):

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

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

```

class FingerFactoryFromService(protocol.ServerFactory):

    implements(IFingerFactory)

    protocol = FingerProtocol

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

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

components.registerAdapter(FingerFactoryFromService,
                           IFingerService,
                           IFingerFactory)

class FingerSetterProtocol(basic.LineReceiver):

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

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

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

class IFingerSetterFactory(Interface):

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

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

class FingerSetterFactoryFromService(protocol.ServerFactory):

    implements(IFingerSetterFactory)

    protocol = FingerSetterProtocol

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

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

components.registerAdapter(FingerSetterFactoryFromService,
                           IFingerSetterService,
                           IFingerSetterFactory)

class IRCReplyBot(irc.IRCClient):

```

```

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

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

class IIRCClientFactory(Interface):

    """
    @ivar nickname
    """

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

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

class IRCCClientFactoryFromService(protocol.ClientFactory):

    implements(IIRCCClientFactory)

    protocol = IRCReplyBot
    nickname = None

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

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

components.registerAdapter(IRCCClientFactoryFromService,
                           IFingerService,
                           IIRCCClientFactory)

class UserStatusTree(resource.Resource):

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

        # add a specific child for the path "RPC2"
        self.putChild("RPC2", UserStatusXR(self.service))

        # need to do this for resources at the root of the site
        self.putChild("", self)

    def _cb_render_GET(self, users, request):
        userOutput = ''.join(["<li><a href=\"%s\">%s</a></li>" % (user, user)

```



```

        for user in users])
    request.write("""
        <html><head><title>Users</title></head><body>
        <h1>Users</h1>
        <ul>
        %s
        </ul></body></html>""" % userOutput)
    request.finish()

def render_GET(self, request):
    d = self.service.getUsers()
    d.addCallback(self._cb_render_GET, request)

    # signal that the rendering is not complete
    return server.NOT_DONE_YET

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

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

class UserStatus(resource.Resource):

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

    def _cb_render_GET(self, status, request):
        request.write("""<html><head><title>%s</title></head>
        <body><h1>%s</h1>
        <p>%s</p>
        </body></html>""" % (self.user, self.user, status))
        request.finish()

    def render_GET(self, request):
        d = self.service.getUser(self.user)
        d.addCallback(self._cb_render_GET, request)

        # signal that the rendering is not complete
        return server.NOT_DONE_YET

class UserStatusXR(xmlrpc.XMLRPC):

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

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

    def xmlrpc_getUsers(self):
        return self.service.getUsers()

class FingerService(service.Service):

```

```

implements(IFingerService)

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

def _read(self):
    self.users = {}
    for line in file(self.filename):
        user, status = line.split(':', 1)
        user = user.strip()
        status = status.strip()
        self.users[user] = status
    self.call = reactor.callLater(30, self._read)

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

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

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

```

Source listing — *finger20.py*

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

2.12.1 Introduction

This is the seventh part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 25).

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

2.12.2 Use Perspective Broker

We add support for perspective broker, Twisted's native remote object protocol. Now, Twisted clients will not have to go through XML-RPCish contortions to get information about users.

```

# Do everything properly, and componentize
from twisted.application import internet, service
from twisted.internet import protocol, reactor, defer
from twisted.words.protocols import irc

```

```

from twisted.protocols import basic
from twisted.python import components
from twisted.web import resource, server, static, xmlrpc, microdom
from twisted.spread import pb
from zope.interface import Interface, implements
import cgi

class IFingerService(Interface):

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

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

class IFingerSetterService(Interface):

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

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

class FingerProtocol(basic.LineReceiver):

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

class IFingerFactory(Interface):

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

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

class FingerFactoryFromService(protocol.ServerFactory):

    implements(IFingerFactory)

    protocol = FingerProtocol

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

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

components.registerAdapter(FingerFactoryFromService,
                           IFingerService,

```

```

        IFingerFactory)

class FingerSetterProtocol(basic.LineReceiver):

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

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

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

class IFingerSetterFactory(Interface):

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

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

class FingerSetterFactoryFromService(protocol.ServerFactory):

    implements(IFingerSetterFactory)

    protocol = FingerSetterProtocol

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

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

components.registerAdapter(FingerSetterFactoryFromService,
                           IFingerSetterService,
                           IFingerSetterFactory)

class IRCReplyBot(irc.IRCClient):

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

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

class IIRCClientFactory(Interface):

```

```

    """
    @ivar nickname
    """

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

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

class IRCCliientFactoryFromService(protocol.ClientFactory):

    implements(IIRCCliientFactory)

    protocol = IRCReplyBot
    nickname = None

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

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

components.registerAdapter(IRCCliientFactoryFromService,
                           IFingerService,
                           IIRCCliientFactory)

class UserStatusTree(resource.Resource):

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

        # add a specific child for the path "RPC2"
        self.putChild("RPC2", UserStatusXR(self.service))

        # need to do this for resources at the root of the site
        self.putChild("", self)

    def _cb_render_GET(self, users, request):
        userOutput = ''.join(["<li><a href=\"%s\">%s</a></li>" % (user, user)
                               for user in users])
        request.write("""
            <html><head><title>Users</title></head><body>
            <h1>Users</h1>
            <ul>
            %s
            </ul></body></html>""" % userOutput)
        request.finish()

    def render_GET(self, request):
        d = self.service.getUsers()
        d.addCallback(self._cb_render_GET, request)

        # signal that the rendering is not complete

```

```

        return server.NOT_DONE_YET

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

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

class UserStatus(resource.Resource):

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

    def _cb_render_GET(self, status, request):
        request.write("""<html><head><title>%s</title></head>
<body><h1>%s</h1>
<p>%s</p>
</body></html>""") % (self.user, self.user, status))
        request.finish()

    def render_GET(self, request):
        d = self.service.getUser(self.user)
        d.addCallback(self._cb_render_GET, request)

        # signal that the rendering is not complete
        return server.NOT_DONE_YET

class UserStatusXR(xmlrpc.XMLRPC):

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

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

    def xmlrpc_getUsers(self):
        return self.service.getUsers()

class IPerspectiveFinger(Interface):

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

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

class PerspectiveFingerFromService(pb.Root):

    implements(IPerspectiveFinger)

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

    def remote_getUser(self, username):

```

```

        return self.service.getUser(username)

    def remote_getUsers(self):
        return self.service.getUsers()

components.registerAdapter(PerspectiveFingerFromService,
                           IFingerService,
                           IPerspectiveFinger)

class FingerService(service.Service):

    implements(IFingerService)

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

    def _read(self):
        self.users = {}
        for line in file(self.filename):
            user, status = line.split(':', 1)
            user = user.strip()
            status = status.strip()
            self.users[user] = status
        self.call = reactor.callLater(30, self._read)

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

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

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

```

Source listing — *finger21.py*

A simple client to test the perspective broker finger:

```

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

from twisted.spread import pb

```

```

from twisted.internet import reactor

def gotObject(object):
    print "got object:", object
    object.callRemote("getUser", "moshez").addCallback(gotData)
# or
# object.callRemote("getUsers").addCallback(gotData)

def gotData(data):
    print 'server sent:', data
    reactor.stop()

def gotNoObject(reason):
    print "no object:", reason
    reactor.stop()

factory = pb.PBClientFactory()
reactor.connectTCP("127.0.0.1", 8889, factory)
factory.getRootObject().addCallbacks(gotObject, gotNoObject)
reactor.run()

```

Source listing — *fingerPBclient.py*

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

2.13.1 Introduction

This is the eighth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 25).

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

2.13.2 Support HTTPS

All we need to do to code an HTTPS site is just write a context factory (in this case, which loads the certificate from a certain file) and then use the `twisted.application.internet.SSLServer` method. Note that one factory (in this case, a site) can listen on multiple ports with multiple protocols.

```

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

class IFingerService(Interface):

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

    def getUsers():

```



```

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

class IFingerSetterService(Interface):

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

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

class FingerProtocol(basic.LineReceiver):

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

class IFingerFactory(Interface):

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

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

class FingerFactoryFromService(protocol.ServerFactory):

    implements(IFingerFactory)

    protocol = FingerProtocol

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

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

components.registerAdapter(FingerFactoryFromService,
                           IFingerService,
                           IFingerFactory)

class FingerSetterProtocol(basic.LineReceiver):

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

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

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

```

```

class IFingerSetterFactory(Interface):

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

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

class FingerSetterFactoryFromService(protocol.ServerFactory):

    implements(IFingerSetterFactory)

    protocol = FingerSetterProtocol

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

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

components.registerAdapter(FingerSetterFactoryFromService,
                           IFingerSetterService,
                           IFingerSetterFactory)

class IRCReplyBot(irc.IRCClient):

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

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

class IIRCClientFactory(Interface):

    """
    @ivar nickname
    """

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

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

class IRCClientFactoryFromService(protocol.ClientFactory):

```

```

implements(IIRCClientFactory)

protocol = IRCReplyBot
nickname = None

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

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

components.registerAdapter(IRCCClientFactoryFromService,
                           IFingerService,
                           IIRCCClientFactory)

class UserStatusTree(resource.Resource):

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

        # add a specific child for the path "RPC2"
        self.putChild("RPC2", UserStatusXR(self.service))

        # need to do this for resources at the root of the site
        self.putChild("", self)

    def _cb_render_GET(self, users, request):
        userOutput = ''.join(["<li><a href=\"%s\">%s</a></li>" % (user, user)
                               for user in users])
        request.write("""
            <html><head><title>Users</title></head><body>
            <h1>Users</h1>
            <ul>
            %s
            </ul></body></html>""" % userOutput)
        request.finish()

    def render_GET(self, request):
        d = self.service.getUsers()
        d.addCallback(self._cb_render_GET, request)

        # signal that the rendering is not complete
        return server.NOT_DONE_YET

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

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

class UserStatus(resource.Resource):

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

```

```

def _cb_render_GET(self, status, request):
    request.write("""<html><head><title>%s</title></head>
<body><h1>%s</h1>
<p>%s</p>
</body></html>""") % (self.user, self.user, status))
    request.finish()

def render_GET(self, request):
    d = self.service.getUser(self.user)
    d.addCallback(self._cb_render_GET, request)

    # signal that the rendering is not complete
    return server.NOT_DONE_YET

class UserStatusXR(xmlrpc.XMLRPC):

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

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

    def xmlrpc_getUsers(self):
        return self.service.getUsers()

class IPerspectiveFinger(Interface):

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

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

class PerspectiveFingerFromService(pb.Root):

    implements(IPerspectiveFinger)

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

    def remote_getUser(self, username):
        return self.service.getUser(username)

    def remote_getUsers(self):
        return self.service.getUsers()

components.registerAdapter(PerspectiveFingerFromService,
                           IFingerService,
                           IPerspectiveFinger)

class FingerService(service.Service):

    implements(IFingerService)

```

```

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

def _read(self):
    self.users = {}
    for line in file(self.filename):
        user, status = line.split(':', 1)
        user = user.strip()
        status = status.strip()
        self.users[user] = status
    self.call = reactor.callLater(30, self._read)

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

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

class ServerContextFactory:

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

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

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

```

2.14 The Evolution of Finger: a Twisted finger client

2.14.1 Introduction

This is the ninth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 25).

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

2.14.2 Finger Proxy

Writing new clients with Twisted is much like writing new servers. We implement the protocol, which just gathers up all the data, and give it to the factory. The factory keeps a deferred which is triggered if the connection either fails or succeeds. When we use the client, we first make sure the deferred will never fail, by producing a message in that case. Implementing a wrapper around client which just returns the deferred is a common pattern. While less flexible than using the factory directly, it's also more convenient.

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

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

class IFingerService(Interface):

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

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

class IFingerFactory(Interface):

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

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

class FingerProtocol(basic.LineReceiver):

    def lineReceived(self, user):
        d = self.factory.getUser(user)
        d.addErrback(catchError)
        def writeValue(value):
            self.transport.write(value)
            self.transportloseConnection()
        d.addCallback(writeValue)

class FingerFactoryFromService(protocol.ClientFactory):
```

```

    implements(IFingerFactory)

    protocol = FingerProtocol

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

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

components.registerAdapter(FingerFactoryFromService,
                           IFingerService,
                           IFingerFactory)

class FingerClient(protocol.Protocol):

    def connectionMade(self):
        self.transport.write(self.factory.user+"\r\n")
        self.buf = []

    def dataReceived(self, data):
        self.buf.append(data)

    def connectionLost(self, reason):
        self.factory.gotData(''.join(self.buf))

class FingerClientFactory(protocol.ClientFactory):

    protocol = FingerClient

    def __init__(self, user):
        self.user = user
        self.d = defer.Deferred()

    def clientConnectionFailed(self, _, reason):
        self.d.errback(reason)

    def gotData(self, data):
        self.d.callback(data)

def finger(user, host, port=79):
    f = FingerClientFactory(user)
    reactor.connectTCP(host, port, f)
    return f.d

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

    def getUser(self, user):
        try:
            user, host = user.split('@', 1)
        except:
            user = user.strip()
            host = '127.0.0.1'

```

```

        ret = finger(user, host)
        ret.addErrback(lambda _: "Could not connect to remote host")
        return ret

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

application = service.Application('finger', uid=1, gid=1)
f = ProxyFingerService()
internet.TCPServer(7779, IFingerFactory(f)).setServiceParent(
    service.IServiceCollection(application))

```

Source listing — *fingerproxy.py*

2.15 The Evolution of Finger: making a finger library

2.15.1 Introduction

This is the tenth part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 25).

In this part, we separate the application code that launches a finger service from the library code which defines a finger service, placing the application in a Twisted Application Configuration (.tac) file. We also move configuration (such as HTML templates) into separate files.

2.15.2 Organization

Now this code, while quite modular and well-designed, isn't properly organized. Everything above the `application =` belongs in a module, and the HTML templates all belong in separate files.

We can use the `templateFile` and `templateDirectory` attributes to indicate what HTML template file to use for each Page, and where to look for it.

```

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

import finger

from twisted.internet import protocol, reactor, defer
from twisted.spread import pb
from twisted.web import resource, server
from twisted.application import internet, service, strports
from twisted.python import log

application = service.Application('finger', uid=1, gid=1)
f = finger.FingerService('/etc/users')
serviceCollection = service.IServiceCollection(application)
internet.TCPServer(79, finger.IFingerFactory(f)
    ).setServiceParent(serviceCollection)

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

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

i = finger.IIRCClientFactory(f)

```



```

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

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

```

Source listing — *organized-finger.tac*

Note that our program is now quite separated. We have:

- Code (in the module)
- Configuration (file above)
- Presentation (templates)
- Content (/etc/users)
- Deployment (twistd)

Prototypes don't need this level of separation, so our earlier examples all bunched together. However, real applications do. Thankfully, if we write our code correctly, it is easy to achieve a good separation of parts.

2.15.3 Easy Configuration

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

```

# Easy configuration
# makeService from finger module

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

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

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

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

```

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

return s
```

Source listing — *finger_config.py*

And we can write simpler files now:

```
# simple-finger.tac
# eg: twisted -ny simple-finger.tac

from twisted.application import service

import finger

options = { 'file': '/etc/users',
            'templates': '/usr/share/finger/templates',
            'ircnick': 'fingerbot',
            'ircserver': 'irc.freenode.net',
            'pbport': 8889,
            'ssl': 'ssl=0' }

ser = finger.makeService(options)
application = service.Application('finger', uid=1, gid=1)
ser.setServiceParent(service.IServiceCollection(application))
```

Source listing — *simple-finger.tac*

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

Note: the finger *user* still has ultimate power: he can use `makeService`, or he can use the lower-level interface if he has specific needs (maybe an IRC server on some other port? maybe we want the non-SSL webserver to listen only locally? etc. etc.) This is an important design principle: never force a layer of abstraction: allow usage of layers of abstractions.

The pasta theory of design:

- Spaghetti: each piece of code interacts with every other piece of code [can be implemented with GOTO, functions, objects]
- Lasagna: code has carefully designed layers. Each layer is, in theory independent. However low-level layers usually cannot be used easily, and high-level layers depend on low-level layers.
- Ravioli: each part of the code is useful by itself. There is a thin layer of interfaces between various parts [the sauce]. Each part can be usefully be used elsewhere.
- ...but sometimes, the user just wants to order “Ravioli”, so one coarse-grain easily definable layer of abstraction on top of it all can be useful.

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

2.16.1 Introduction

This is the eleventh part of the Twisted tutorial *Twisted from Scratch, or The Evolution of Finger* (page 25).

In this part, we make it easier for non-programmers to configure a finger server, and show how to package it in the .deb and RPM package formats.

2.16.2 Plugins

So far, the user had to be somewhat of a programmer to be able to configure stuff. Maybe we can eliminate even that? Move old code to `finger/_init_.py` and...

Full source code for finger module here:

```
# finger.py module

from zope.interface import Interface, implements

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

class IFingerService(Interface):

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

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

class IFingerSetterService(Interface):

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

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

class FingerProtocol(basic.LineReceiver):

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

class IFingerFactory(Interface):

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

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

```

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

    protocol = FingerProtocol

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

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

components.registerAdapter(FingerFactoryFromService,
                           IFingerService,
                           IFingerFactory)

class FingerSetterProtocol(basic.LineReceiver):

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

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

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

class IFingerSetterFactory(Interface):

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

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

class FingerSetterFactoryFromService(protocol.ServerFactory):

    implements(IFingerSetterFactory)

    protocol = FingerSetterProtocol

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

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

components.registerAdapter(FingerSetterFactoryFromService,
                           IFingerSetterService,
                           IFingerSetterFactory)

class IRCReplyBot(irc.IRCClient):

```

```

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

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

class IIRCClientFactory(Interface):

    """
    @ivar nickname
    """

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

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

class IRCClientFactoryFromService(protocol.ClientFactory):

    implements(IIRCClientFactory)

    protocol = IRCReplyBot
    nickname = None

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

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

components.registerAdapter(IRCClientFactoryFromService,
                           IFingerService,
                           IIRCClientFactory)

class UsersModel(model.MethodModel):

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

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

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

class UserStatusTree(page.Page):

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

```

```

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

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

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

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

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

class UserStatus(page.Page):

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

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

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

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

class UserStatusXR(xmlrpc.XMLRPC):

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

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

    def xmlrpc_getUsers(self):
        return self.service.getUsers()

class IPerspectiveFinger(Interface):

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

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

class PerspectiveFingerFromService(pb.Root):

```

```

implements(IPerspectiveFinger)

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

def remote_getUser(self, username):
    return self.service.getUser(username)

def remote_getUsers(self):
    return self.service.getUsers()

components.registerAdapter(PerspectiveFingerFromService,
                           IFingerService,
                           IPerspectiveFinger)

class FingerService(service.Service):

    implements(IFingerService)

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

    def _read(self):
        self.users = {}
        for line in file(self.filename):
            user, status = line.split(':', 1)
            user = user.strip()
            status = status.strip()
            self.users[user] = status
        self.call = reactor.callLater(30, self._read)

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

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

class ServerContextFactory:

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

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

# Easy configuration

```

```

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

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

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

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

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

    return s

```

finger module — *finger.py*

```

# finger/tap.py
from twisted.application import internet, service
from twisted.internet import interfaces
from twisted.python import usage
import finger

class Options(usage.Options):

    optParameters = [
        ['file', 'f', '/etc/users'],
        ['templates', 't', '/usr/share/finger/templates'],
        ['ircnick', 'n', 'fingerbot'],
        ['ircserver', None, 'irc.freenode.net'],
        ['pbport', 'p', 8889],
    ]

    optFlags = [['ssl', 's']]

```



```
def makeService(config):
    return finger.makeService(config)
```

finger/tap.py — tap.py

And register it all:

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

finger/plugins.tml — plugins.tml

And now, the following works

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

2.16.3 OS Integration

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

Debian

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

Red Hat / Mandrake

```
% tap2rpm --type=python finger.tac #[maybe other options needed]
% sudo rpm -i .build/*.rpm
```

Will properly register the tap/tac, init.d scripts, etc. for the given file.

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

Chapter 3

Low-Level Twisted

3.1 Reactor Overview

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

3.1.1 Reactor Basics

The reactor is the core of the event loop within Twisted – the loop which drives applications using Twisted. The reactor provides basic interfaces to a number of services, including network communications, threading, and event dispatching.

For information about using the reactor and the Twisted event loop, see:

- the event dispatching howtos: *Scheduling* (page 133) and *Using Deferreds* (page 99);
- the communication howtos: *TCP servers* (page 13), *TCP clients* (page 17), *UDP networking* (page 90) and *Using processes* (page 92); and
- *Using threads* (page 133).

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 135).

Twisted applications can use the interfaces in `twisted.application.service` to configure and run the application instead of using boilerplate reactor code. See *Using Application* (page 155) for an introduction to Application.

3.1.2 Using the reactor object

You can get to the reactor object using the following code:

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

The reactor usually implements a set of interfaces, but depending on the chosen reactor and the platform, some of the interfaces may not be implemented:

- `IReactorCore`: Core (required) functionality.
- `IReactorFDSet`: Use `FileDescriptor` objects.
- `IReactorProcess`: Process management. Read the *Using Processes* (page 92) document for more information.
- `IReactorSSL`: SSL networking support.
- `IReactorTCP`: TCP networking support. More information can be found in the *Writing Servers* (page 13) and *Writing Clients* (page 17) documents.

- `IReactorThreads`: Threading use and management. More information can be found within *Threading In Twisted* (page 133).
- `IReactorTime`: Scheduling interface. More information can be found within *Scheduling Tasks* (page 133).
- `IReactorUDP`: UDP networking support. More information can be found within *UDP Networking* (this page).
- `IReactorUNIX`: UNIX socket support.

3.2 UDP Networking

3.2.1 Overview

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

Since there are no multiple connections, we only use a single object, a protocol, for each UDP socket. We then use the reactor to connect this protocol to a UDP transport, using the `twisted.internet.interfaces.IReactorUDP` reactor API.

3.2.2 DatagramProtocol

At the base, the place where you actually implement the protocol parsing and handling, is the `DatagramProtocol` class. This class will usually be descended 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 registered 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.IUDPTTransport` interface. Notice that the `host` argument should be an IP, not a hostname. If you only have the hostname use `reactor.resolve()` to resolve the address (see `twisted.internet.interfaces.IReactorCore.resolve`).

3.2.3 Connected UDP

A connected UDP socket is slightly different from a standard one - it can only send and receive datagrams to/from a single address, but this does not in any way imply a connection. Datagrams may still arrive in any order, and the port on the other side may have no one listening. The benefit of the connected UDP socket is that it *may* provide notification of undelivered packages. This depends on many factors, almost all of which are out of the control of the application, but it still presents certain benefits which occasionally make it useful.

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

```

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

class Helloer(DatagramProtocol):

    def startProtocol(self):
        self.transport.connect("192.168.1.1", 1234)
        print "we can only send to %s now" % str((host, port))
        self.transport.write("hello") # no need for address

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

# Possibly invoked if there is no server listening on the
# address to which we are sending.
def connectionRefused(self):
    print "Noone listening"

# 0 means any port, we don't care in this case
reactor.listenUDP(0, Helloer())
reactor.run()

```

Note that `connect()`, like `write()` will only accept IP addresses, not unresolved domain names. To obtain the IP of a domain name use `reactor.resolve()`, e.g.:

```

from twisted.internet import reactor

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

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

```

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

3.2.4 Multicast UDP

A multicast UDP socket can send and receive datagrams from multiple clients. The interesting and useful feature of the multicast is that a client can contact multiple servers with a single packet, without knowing the specific IP of any of the hosts.

```

from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor
from twisted.application.internet import MulticastServer

class MulticastServerUDP(DatagramProtocol):
    def startProtocol(self):
        print 'Started Listening'
        # Join a specific multicast group, which is the IP we will respond to
        self.transport.joinGroup('224.0.0.1')

    def datagramReceived(self, datagram, address):
        # The uniqueID check is to ensure we only service requests from
        # ourselves
        if datagram == 'UniqueID':
            print "Server Received:" + repr(datagram)
            self.transport.write("data", address)

# Note that the join function is picky about having a unique object

```

```
# on which to call join. To avoid using startProtocol, the following is
# sufficient:
# reactor.listenMulticast(8005, MulticastServerUDP()).join('224.0.0.1')

# Listen for multicast on 224.0.0.1:8005
reactor.listenMulticast(8005, MulticastServerUDP())
reactor.run()
```

Source listing — *MulticastServer.py*

The server protocol is very simple, and closely resembles a normal listenUDP implementation. The main difference is that instead of listenUDP, listenMulticast is called with a specified port number. The server must also call joinGroup to specify on which multicast IP address it will service requests. Another item of interest is the contents of the datagram. Many different applications use multicast as a way of device discovery, which leads to an abundance of packets flying around. Checking the payload can ensure that we only service requests from our specific clients.

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

class MulticastClientUDP(DatagramProtocol):

    def datagramReceived(self, datagram, address):
        print "Received:" + repr(datagram)

# Send multicast on 224.0.0.1:8005, on our dynamically allocated port
reactor.listenUDP(0, MulticastClientUDP()).write('UniqueID',
                                                  ('224.0.0.1', 8005))
reactor.run()
```

MulticastServer.py — *MulticastClient.py*

This is a mirror implementation of a standard UDP client. The only difference is that the destination IP is the multicast address. This datagram will be distributed to every server listening on 224.0.0.1 and port 8005. Note that the client port is specified as 0, as we have no need to keep track of what port the client is listening on.

3.2.5 Acknowledgements

Thank you to all contributors to this document, including:

- Kyle Robertson, author of the explanation and examples of multicast

3.3 Using Processes

3.3.1 Overview

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

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

3.3.2 Running Another Process

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

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

```
mypp = MyProcessProtocol()
reactor.spawnProcess(processProtocol, executable, args=[program, arg1, arg2],
                    env={'HOME': os.environ['HOME']}, path,
                    uid, gid, usePTY, childFDs)
```

- `processProtocol` should be an instance of a subclass of `twisted.internet.protocol.ProcessProtocol`. The interface is described below.
- `executable` is the full path of the program to run. It will be connected to `processProtocol`.
- `args` is a list of command line arguments to be passed to the process. `args[0]` should be the name of the process.
- `env` is a dictionary containing the environment to pass through to the process.
- `path` is the directory to run the process in. The child will switch to the given directory just before starting the new program. The default is to stay in the current directory.
- `uid` and `gid` are the user ID and group ID to run the subprocess as. Of course, changing identities will be more likely to succeed if you start as root.
- `usePTY` specifies whether the child process should be run with a pty, or if it should just get a pair of pipes. Interactive programs (where you don't know when it may read or write) need to be run with ptys.
- `childFDs` lets you specify how the child's file descriptors should be set up. Each key is a file descriptor number (an integer) as seen by the child. 0, 1, and 2 are usually stdin, stdout, and stderr, but some programs may be instructed to use additional fds through command-line arguments or environment variables. Each value is either an integer specifying one of the parent's current file descriptors, the string "r" which creates a pipe that the parent can read from, or the string "w" which creates a pipe that the parent can write to. If `childFDs` is not provided, a default is used which creates the usual stdin-writer, stdout-reader, and stderr-reader pipes.

`args` and `env` have empty default values, but many programs depend upon them to be set correctly. At the very least, `args[0]` should probably be the same as `executable`. If you just provide `os.environ` for `env`, the child program will inherit the environment from the current process, which is usually the civilized thing to do (unless you want to explicitly clean the environment as a security precaution). The default is to give an empty `env` to the child.

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

3.3.3 Writing a ProcessProtocol

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

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

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

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

```
def connectionMade(self):
    self.transport.write(self.text)
    self.transport.closeStdin()
```

At this point, the process has received the data, and it's time for us to read the results. Instead of being received in `dataReceived`, data from standard output is received in `outReceived`. This is to distinguish it from data on standard error.

```
def outReceived(self, data):
    fieldLength = len(data) / 3
    lines = int(data[:fieldLength])
    words = int(data[fieldLength:fieldLength*2])
    chars = int(data[fieldLength*2:])
    self.transportloseConnection()
    self.receiveCounts(lines, words, chars)
```

Now, the process has parsed the output, and ended the connection to the process. Then it sends the results on to the final method, `receiveCounts`. This is for users of the class to override, so as to do other things with the data. For our demonstration, we will just print the results.

```
def receiveCounts(self, lines, words, chars):
    print 'Received counts from wc.'
    print 'Lines:', lines
    print 'Words:', words
    print 'Characters:', chars
```

We're done! To use our `WCProcessProtocol`, we create an instance, and pass it to `spawnProcess`.

```
from twisted.internet import reactor
wcProcess = WCProcessProtocol("accessing protocols through Twisted is fun!\n")
reactor.spawnProcess(wcProcess, 'wc', ['wc'])
reactor.run()
```

3.3.4 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.transportloseConnection()`.
- `.outConnectionLost`: This is called when the program closes its stdout pipe. This usually happens when the program terminates.
- `.errConnectionLost`: Same as `outConnectionLost`, but for stderr instead of stdout.
- `.processEnded(status)`: This is called when the child process has been reaped, and receives information about the process' exit status. The status is passed in the form of a `Failure` instance, created with a `.value` that either holds a `ProcessDone` object if the process terminated normally (it died of natural causes instead of receiving a signal, and if the exit code was 0), or a `ProcessTerminated` object (with an `.exitCode`

attribute) if something went wrong. This scheme may seem a bit weird, but I trust that it proves useful when dealing with exceptions that occur in asynchronous code.

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

The base-class definitions of these functions are all no-ops. This will result in all `stdout` and `stderr` being thrown away. Note that it is important for data you don't care about to be thrown away: if the pipe were not read, the child process would eventually block as it tried to write to a full pipe.

3.3.5 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.closeConnection`: Close all three pipes.
- `self.transport.signalProcess('KILL')`: Kill the child process. This will eventually result in `processEnded` being called.

3.3.6 Verbose Example

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

```
#!/usr/bin/python

from twisted.internet import protocol
from twisted.internet import reactor
import re

class MyPP(protocol.ProcessProtocol):
    def __init__(self, verses):
        self.verses = verses
        self.data = ""
    def connectionMade(self):
        print "connectionMade!"
        for i in range(self.verses):
            self.transport.write("Aleph-null bottles of beer on the wall,\n" +
                                "Aleph-null bottles of beer,\n" +
                                "Take one down and pass it around,\n" +
                                "Aleph-null bottles of beer on the wall.\n")
        self.transport.closeStdin() # tell them we're done
    def outReceived(self, data):
        print "outReceived! with %d bytes!" % len(data)
        self.data = self.data + data
    def errReceived(self, data):
        print "errReceived! with %d bytes!" % len(data)
```



```

def inConnectionLost(self):
    print "inConnectionLost! stdin is closed! (we probably did it)"
def outConnectionLost(self):
    print "outConnectionLost! The child closed their stdout!"
    # now is the time to examine what they wrote
    #print "I saw them write:", self.data
    (dummy, lines, words, chars, file) = re.split(r'\s+', self.data)
    print "I saw %s lines" % lines
def errConnectionLost(self):
    print "errConnectionLost! The child closed their stderr."
def processEnded(self, status_object):
    print "processEnded, status %d" % status_object.value.exitCode
    print "quitting"
    reactor.stop()

pp = MyPP(10)
reactor.spawnProcess(pp, "wc", ["wc"], {})
reactor.run()

```

Source listing — *process.py*

The exact output of this program depends upon the relative timing of some un-synchronized events. In particular, the program may observe the child process close its stderr pipe before or after it reads data from the stdout pipe. One possible transcript would look like this:

```

% ./process.py
connectionMade!
inConnectionLost! stdin is closed! (we probably did it)
errConnectionLost! The child closed their stderr.
outReceived! with 24 bytes!
outConnectionLost! The child closed their stdout!
I saw 40 lines
processEnded, status 0
quitting
Main loop terminated.
%

```

3.3.7 Doing it the Easy Way

Frequently, one just needs a simple way to get all the output from a program. In the blocking world, you might use `commands.getoutput` from the standard library, but using that in an event-driven program will cause everything else to stall until the command finishes. (in addition, the `SIGCHLD` handler used by that function does not play well with Twisted's own signal handling). For these cases, the `twisted.internet.utils.getProcessOutput` function can be used. Here is a simple example:

```

from twisted.internet import protocol, utils, reactor
from twisted.python import failure
from cStringIO import StringIO

class FortuneQuoter(protocol.Protocol):

    fortune = '/usr/games/fortune'

    def connectionMade(self):
        output = utils.getProcessOutput(self.fortune)
        output.addCallbacks(self.writeResponse, self.noResponse)

    def writeResponse(self, resp):

```

```

        self.transport.write(resp)
        self.transport.looseConnection()

    def noResponse(self, err):
        self.transport.looseConnection()

if __name__ == '__main__':
    f = protocol.Factory()
    f.protocol = FortuneQuoter
    reactor.listenTCP(10999, f)
    reactor.run()

```

Source listing — *quotes.py*

If you only need the final exit code (like `commands.getstatusoutput(cmd)[0]`), the `twisted.internet.utils.getProcessValue` function is useful. Here is an example:

```

from twisted.internet import utils, reactor

def printTrueValue(val):
    print "/bin/true exits with rc=%d" % val
    output = utils.getProcessValue('/bin/false')
    output.addCallback(printFalseValue)

def printFalseValue(val):
    print "/bin/false exits with rc=%d" % val
    reactor.stop()

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

```

Source listing — *trueandfalse.py*

3.3.8 Mapping File Descriptors

“stdin”, “stdout”, and “stderr” are just conventions. Programs which operate as filters generally accept input on `fd0`, write their output on `fd1`, and emit error messages on `fd2`. This is common enough that the standard C library provides macros like “`stdin`” to mean `fd0`, and shells interpret the pipe character “`—`” to mean “redirect `fd1` from one command into `fd0` of the next command”.

But these are just conventions, and programs are free to use additional file descriptors or even ignore the standard three entirely. The “`childFDs`” argument allows you to specify exactly what kind of files descriptors the child process should be given.

Each child FD can be put into one of three states:

- Mapped to a parent FD: this causes the child’s reads and writes to come from or go to the same source/destination as the parent.
- Feeding into a pipe which can be read by the parent.
- Feeding from a pipe which the parent writes into.

Mapping the child FDs to the parent’s is very commonly used to send the child’s `stderr` output to the same place as the parent’s. When you run a program from the shell, it will typically leave `fds 0, 1, and 2` mapped to the shell’s `0, 1, and 2`, allowing you to see the child program’s output on the same terminal you used to launch the child. Likewise, `inetd` will typically map both `stdin` and `stdout` to the network socket, and may map `stderr` to the same socket or to some

kind of logging mechanism. This allows the child program to be implemented with no knowledge of the network: it merely speaks its protocol by doing reads on `fd0` and writes on `fd1`.

Feeding into a parent's read pipe is used to gather output from the child, and is by far the most common way of interacting with child processes.

Feeding from a parent's write pipe allows the parent to control the child. Programs like "bc" or "ftp" can be controlled this way, by writing commands into their stdin stream.

The "childFDs" dictionary maps file descriptor numbers (as will be seen by the child process) to one of these three states. To map the fd to one of the parent's fds, simply provide the fd number as the value. To map it to a read pipe, use the string "r" as the value. To map it to a write pipe, use the string "w".

For example, the default mapping sets up the standard stdin/stdout/stderr pipes. It is implemented with the following dictionary:

```
childFDs = { 0: "w", 1: "r", 2: "r" }
```

To launch a process which reads and writes to the same places that the parent python program does, use this:

```
childFDs = { 0: 0, 1: 1, 2: 2 }
```

To write into an additional fd (say it is fd number 4), use this:

```
childFDs = { 0: "w", 1: "r", 2: "r" , 4: "w" }
```

ProcessProtocols with extra file descriptors

When you provide a "childFDs" dictionary with more than the normal three fds, you need additional methods to access those pipes. These methods are more generalized than the `.outReceived` ones described above. In fact, those methods (`outReceived` and `errReceived`) are actually just wrappers left in for compatibility with older code, written before this generalized fd mapping was implemented. The new list of things that can happen to your `ProcessProtocol` is as follows:

- `.connectionMade`: This is called when the program is started.
- `.childDataReceived(childFD, data)`: This is called with data that was received from one of the process' output pipes (i.e. where the `childFDs` value was "r". The actual file number (from the point of view of the child process) is in "childFD". For compatibility, the default implementation of `.dataReceived` dispatches to `.outReceived` or `.errReceived` when "childFD" is 1 or 2.
- `.childConnectionLost(childFD)`: This is called when the reactor notices that one of the process' pipes has been closed. This either means you have just closed down the parent's end of the pipe (with `.transport.closeChildFD`), the child closed the pipe explicitly (sometimes to indicate EOF), or the child process has terminated and the kernel has closed all of its pipes. The "childFD" argument tells you which pipe was closed. Note that you can only find out about file descriptors which were mapped to pipes: when they are mapped to existing fds the parent has no way to notice when they've been closed. For compatibility, the default implementation dispatches to `.inConnectionLost`, `.outConnectionLost`, or `.errConnectionLost`.
- `.processEnded(status)`: This is called when the child process has been reaped, and all pipes have been closed. This insures that all data written by the child prior to its death will be received before `.processEnded` is invoked.

In addition to those methods, there are other methods available to influence the child process:

- `self.transport.writeToChild(childFD, data)`: Stuff some data into an input pipe. `.write` simply writes to `childFD=0`.
- `self.transport.closeChildFD(childFD)`: Close one of the child's pipes. Closing an input pipe is a common way to indicate EOF to the child process. Closing an output pipe is neither very friendly nor very useful.

Examples

GnuPG, the encryption program, can use additional file descriptors to accept a passphrase and emit status output. These are distinct from `stdin` (used to accept the crypttext), `stdout` (used to emit the plaintext), and `stderr` (used to emit human-readable status/warning messages). The passphrase FD reads until the pipe is closed and uses the resulting string to unlock the secret key that performs the actual decryption. The status FD emits machine-parseable status messages to indicate the validity of the signature, which key the message was encrypted to, etc.

gpg accepts command-line arguments to specify what these fds are, and then assumes that they have been opened by the parent before the gpg process is started. It simply performs reads and writes to these fd numbers.

To invoke gpg in decryption/verification mode, you would do something like the following:

```
class GPGProtocol(ProcessProtocol):
    def __init__(self, crypttext):
        self.crypttext = crypttext
        self.plaintext = ""
        self.status = ""
    def connectionMade(self):
        self.transport.writeToChild(3, self.passphrase)
        self.transport.closeChildFD(3)
        self.transport.writeToChild(0, self.crypttext)
        self.transport.closeChildFD(0)
    def childDataReceived(self, childFD, data):
        if childFD == 1: self.plaintext += data
        if childFD == 4: self.status += data
    def processEnded(self, status):
        rc = status.value.exitCode
        if rc == 0:
            self.deferred.callback(self)
        else:
            self.deferred.errback(rc)

def decrypt(crypttext):
    gp = GPGProtocol(crypttext)
    gp.deferred = Deferred()
    cmd = ["gpg", "--decrypt", "--passphrase-fd", "3", "--status-fd", "4",
           "--batch"]
    p = reactor.spawnProcess(gp, cmd[0], cmd, env=None,
                             childFDs={0:"w", 1:"r", 2:2, 3:"w", 4:"r"})
    return gp.deferred
```

In this example, the status output could be parsed after the fact. It could, of course, be parsed on the fly, as it is a simple line-oriented protocol. Methods from `LineReceiver` could be mixed in to make this parsing more convenient.

The `stderr` mapping (“2:2”) used will cause any GPG errors to be emitted by the parent program, just as if those errors had caused in the parent itself. This is sometimes desirable (it roughly corresponds to letting exceptions propagate upwards), especially if you do not expect to encounter errors in the child process and want them to be more visible to the end user. The alternative is to map `stderr` to a read-pipe and handle any such output from within the `ProcessProtocol` (roughly corresponding to catching the exception locally).

3.4 Deferred Reference

This document is a guide to the behaviour of the `twisted.internet.defer.Deferred` object, and to various ways you can use them when they are returned by functions.

This document assumes that you are familiar with the basic principle that the Twisted framework is structured around: asynchronous, callback-based programming, where instead of having blocking code in your program or using threads to run blocking code, you have functions that return immediately and then begin a callback chain when data is available.

See these documents for more information:

- *Asynchronous Programming with Twisted* (page 8)

After reading this document, the reader should expect to be able to deal with most simple APIs in Twisted and Twisted-using code that return Deferreds.

- what sorts of things you can do when you get a Deferred from a function call; and
- how you can write your code to robustly handle errors in Deferred code.

Unless you're already very familiar with asynchronous programming, it's strongly recommended you read the *Deferreds* section (page 9) of the Asynchronous programming document to get an idea of why Deferreds exist.

3.4.1 Callbacks

A `twisted.internet.defer.Deferred` is a promise that a function will at some point have a result. We can attach callback functions to a Deferred, and once it gets a result these callbacks will be called. In addition Deferreds allow the developer to register a callback for an error, with the default behavior of logging the error. The deferred mechanism standardizes the application programmer's interface with all sorts of blocking or delayed operations.

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

```
def getDummyData(x):
    """
    This function is a dummy which simulates a delayed result and
    returns a Deferred which will fire with that result. Don't try too
    hard to understand this.
    """
    d = defer.Deferred()
    # simulate a delayed result by asking the reactor to fire the
    # Deferred in 2 seconds time with the result x * 3
    reactor.callLater(2, d.callback, x * 3)
    return d

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

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

# manually set up the end of the process by asking the reactor to
# stop itself in 4 seconds time
reactor.callLater(4, reactor.stop)
# start up the Twisted reactor (event loop handler) manually
reactor.run()
```

Multiple callbacks

Multiple callbacks can be added to a Deferred. The first callback in the Deferred's callback chain will be called with the result, the second with the result of the first callback, and so on. Why do we need this? Well, consider a Deferred returned by `twisted.enterprise.adbapi` - the result of a SQL query. A web widget might add a callback that converts this result into HTML, and pass the Deferred onwards, where the callback will be used by twisted to return the result to the HTTP client. The callback chain will be bypassed in case of errors or exceptions.

```
from twisted.internet import reactor, defer

class Getter:
```

```

def gotResults(self, x):
    """
    The Deferred mechanism provides a mechanism to signal error
    conditions. In this case, odd numbers are bad.

    This function demonstrates a more complex way of starting
    the callback chain by checking for expected results and
    choosing whether to fire the callback or errback chain
    """
    if x % 2 == 0:
        self.d.callback(x*3)
    else:
        self.d.errback(ValueError("You used an odd number!"))

def _toHTML(self, r):
    """
    This function converts r to HTML.

    It is added to the callback chain by getDummyData in
    order to demonstrate how a callback passes its own result
    to the next callback
    """
    return "Result: %s" % r

def getDummyData(self, x):
    """
    The Deferred mechanism allows for chained callbacks.
    In this example, the output of gotResults is first
    passed through _toHTML on its way to printData.

    Again this function is a dummy, simulating a delayed result
    using callLater, rather than using a real asynchronous
    setup.
    """
    self.d = defer.Deferred()
    # simulate a delayed result by asking the reactor to schedule
    # gotResults in 2 seconds time
    reactor.callLater(2, self.gotResults, x)
    self.d.addCallback(self._toHTML)
    return self.d

def printData(d):
    print d

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

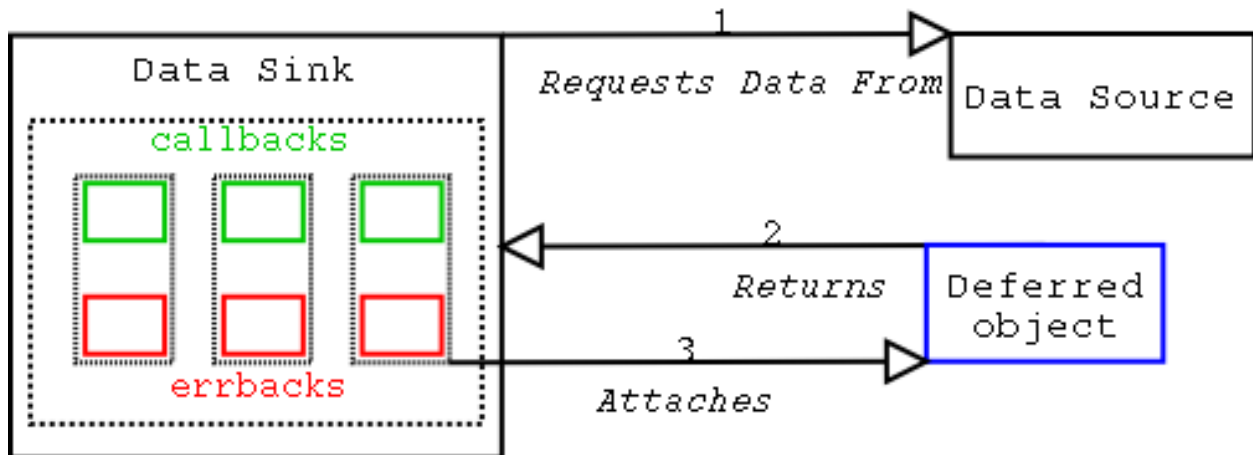
# this series of callbacks and errbacks will print an error message
g = Getter()
d = g.getDummyData(3)
d.addCallback(printData)
d.addErrback(printError)

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

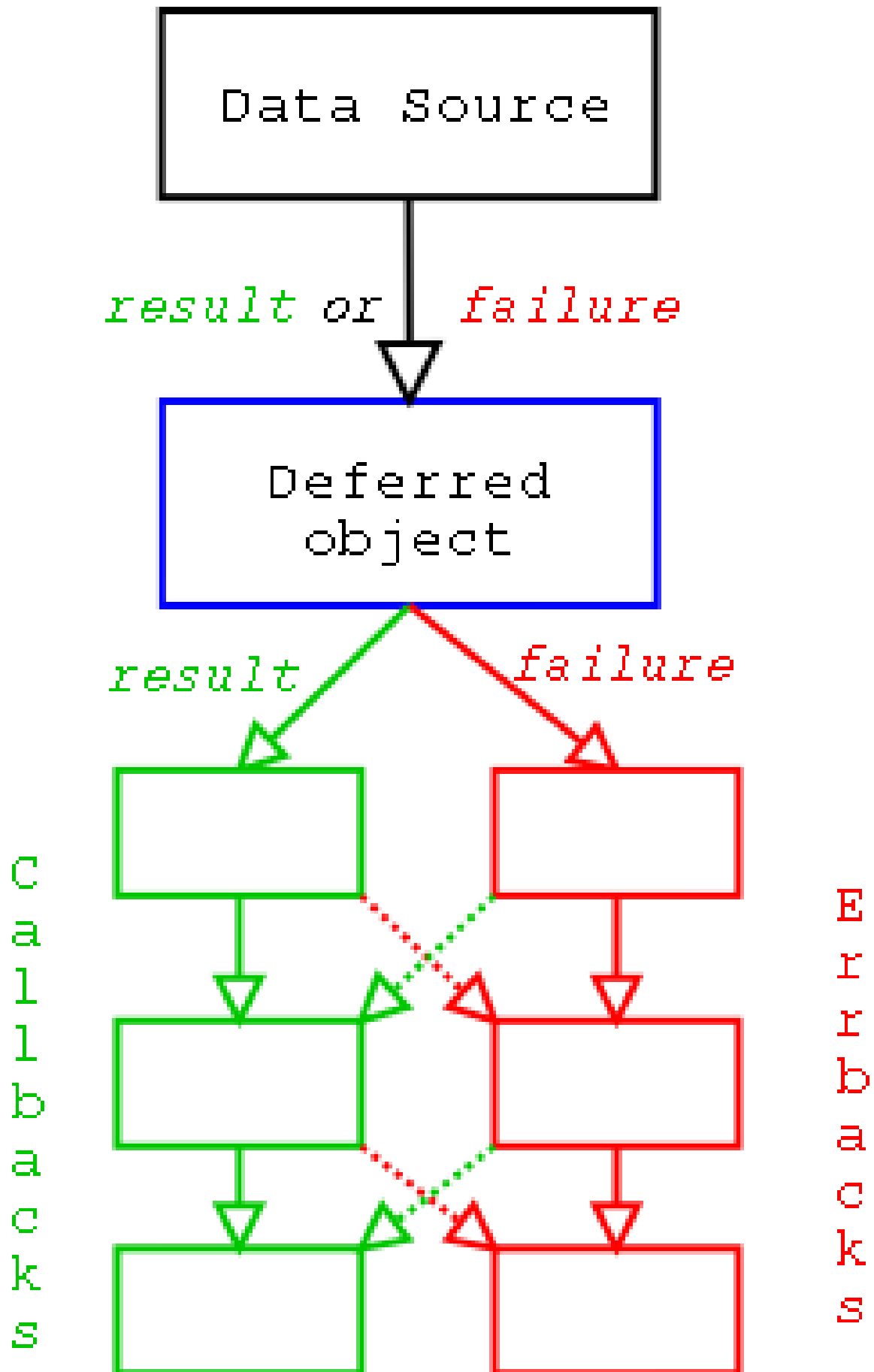
```

```
d.addCallback(printData)
d.addErrback(printError)

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

Visual Explanation

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



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 of `except:` statements.
 - If an errback doesn't raise an exception or return a `twisted.python.failure.Failure` instance, switch to callback.

3.4.2 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 return a `Failure` or raise an exception, then that is passed to the next errback, and so on.

Note: If an errback doesn't return anything, then it effectively returns `None`, meaning that callbacks will continue to be executed after this errback. This may not be what you expect to happen, so be careful. Make sure your errbacks return a `Failure` (probably the one that was passed to it), or a meaningful return value for the next callback.

Also, `twisted.python.failure.Failure` instances have a useful method called `trap`, allowing you to effectively do the equivalent of:

```
try:
    # code that may throw an exception
    cookSpamAndEggs()
except (SpamException, EggException):
    # Handle SpamExceptions and EggExceptions
    ...
```

You do this by:

```
def errorHandler(failure):
    failure.trap(SpamException, EggException)
    # Handle SpamExceptions and EggExceptions

d.addCallback(cookSpamAndEggs)
d.addErrback(errorHandler)
```

If none of arguments passed to `failure.trap` match the error encapsulated in that `Failure`, then it re-raises the error.

There's another potential "gotcha" here. There's a method `twisted.internet.defer.Deferred.addCallbacks` which is similar to, but not exactly the same as, `addCallback` followed by `addErrback`. In particular, consider these two cases:

```
# Case 1
d = getDeferredFromSomewhere()
d.addCallback(callback1)           # A
d.addErrback(errback1)             # B
```

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

# Case 2
d = getDeferredFromSomewhere()
d.addCallbacks(callback1, errback1) # C
d.addCallbacks(callback2, errback2)
```

If an error occurs in `callback1`, then for Case 1 `errback1` will be called with the failure. For Case 2, `errback2` will be called. Be careful with your callbacks and errbacks.

What this means in a practical sense is in Case 1, "A" will handle a success condition from `getDeferredFromSomewhere`, and "B" will handle any errors that occur *from either the upstream source, or that occur in 'A'*. In Case 2, "C"'s `errback1` *will only handle an error condition raised by `getDeferredFromSomewhere`*, it will not do any handling of errors raised in `callback1`.

Unhandled Errors

If a Deferred is garbage-collected with an unhandled error (i.e. it would call the next errback if there was one), then Twisted will write the error's traceback to the log file. This means that you can typically get away with not adding errbacks and still get errors logged. Be careful though; if you keep a reference to the Deferred around, preventing it from being garbage-collected, then you may never see the error (and your callbacks will mysteriously seem to have never been called). If unsure, you should explicitly add an errback after your callbacks, even if all you do is:

```
# Make sure errors get logged
from twisted.python import log
d.addErrback(log.err)
```

3.4.3 Handling either synchronous or asynchronous results

In some applications, there are functions that might be either asynchronous or synchronous. For example, a user authentication function might be able to check in memory whether a user is authenticated, allowing the authentication function to return an immediate result, or it may need to wait on network data, in which case it should return a Deferred to be fired when that data arrives. However, a function that wants to check if a user is authenticated will then need to accept both immediate results *and* Deferreds.

In this example, the library function `authenticateUser` uses the application function `isValidUser` to authenticate a user:

```
def authenticateUser(isValidUser, user):
    if isValidUser(user):
        print "User is authenticated"
    else:
        print "User is not authenticated"
```

However, it assumes that `isValidUser` returns immediately, whereas `isValidUser` may actually authenticate the user asynchronously and return a Deferred. It is possible to adapt this trivial user authentication code to accept either a synchronous `isValidUser` or an asynchronous `isValidUser`, allowing the library to handle either type of function. It is, however, also possible to adapt synchronous functions to return Deferreds. This section describes both alternatives: handling functions that might be synchronous or asynchronous in the library function (`authenticateUser`) or in the application code.

Handling possible Deferreds in the library code

Here is an example of a synchronous user authentication function that might be passed to `authenticateUser`:

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

Source listing — *synch-validation.py*

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

```
from twisted.internet import reactor

def asynchronousIsValidUser(d, user):
    d = Deferred()
    reactor.callLater(2, d.callback, user in ["Alice", "Angus", "Agnes"])
    return d
```

Our original implementation of `authenticateUser` expected `isValidUser` to be synchronous, but now we need to change it to handle both synchronous and asynchronous implementations of `isValidUser`. For this, we use `maybeDeferred` to call `isValidUser`, ensuring that the result of `isValidUser` is a `Deferred`, even if `isValidUser` is a synchronous function:

```
from twisted.internet import defer

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

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

Now `isValidUser` could be either `synchronousIsValidUser` or `asynchronousIsValidUser`.

It is also possible to modify `synchronousIsValidUser` to return a `Deferred`, see *Generating Deferreds* (page 109) for more information.

3.4.4 DeferredList

Sometimes you want to be notified after several different events have all happened, rather than waiting for each one individually. For example, you may want to wait for all the connections in a list to close. `twisted.internet.defer.DeferredList` is the way to do this.

To create a `DeferredList` from multiple `Deferreds`, you simply pass a list of the `Deferreds` you want it to wait for:

```
# Creates a DeferredList
dl = defer.DeferredList([deferred1, deferred2, deferred3])
```

You can now treat the `DeferredList` like an ordinary `Deferred`; you can call `addCallbacks` and so on. The `DeferredList` will call its callback when all the `deferreds` have completed. The callback will be called with a list of the results of the `Deferreds` it contains, like so:

```
def printResult(result):
    print result

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

A standard `DeferredList` will never call `errback`.

Note:

If you want to apply callbacks to the individual `Deferred`s that go into the `DeferredList`, you should be careful about when those callbacks are added. The act of adding a `Deferred` to a `DeferredList` inserts a callback into that `Deferred` (when that callback is run, it checks to see if the `DeferredList` has been completed yet). The important thing to remember is that it is *this callback* which records the value that goes into the result list handed to the `DeferredList`'s callback.

Therefore, if you add a callback to the `Deferred` *after* adding the `Deferred` to the `DeferredList`, the value returned by that callback will not be given to the `DeferredList`'s callback. To avoid confusion, we recommend not adding callbacks to a `Deferred` once it has been used in a `DeferredList`.

```
def printResult(result):
    print result
def addTen(result):
    return result + " ten"

# Deferred gets callback before DeferredList is created
deferred1 = defer.Deferred()
deferred2 = defer.Deferred()
deferred1.addCallback(addTen)
dl = defer.DeferredList([deferred1, deferred2])
dl.addCallback(printResult)
deferred1.callback("one") # fires addTen, checks DeferredList, stores "one ten"
deferred2.callback("two")
# At this point, dl will fire its callback, printing:
#      [(1, 'one ten'), (1, 'two')]
```

```
# Deferred gets callback after DeferredList is created
deferred1 = defer.Deferred()
deferred2 = defer.Deferred()
dl = defer.DeferredList([deferred1, deferred2])
deferred1.addCallback(addTen) # will run *after* DeferredList gets its value
dl.addCallback(printResult)
deferred1.callback("one") # checks DeferredList, stores "one", fires addTen
deferred2.callback("two")
# At this point, dl will fire its callback, printing:
#      [(1, 'one'), (1, 'two')]
```

Other behaviours

`DeferredList` accepts three keyword arguments that modify its behaviour: `fireOnOneCallback`, `fireOnOneErrback` and `consumeErrors`. If `fireOnOneCallback` is set, the `DeferredList` will immediately call its callback as soon as any of its `Deferred`s call their callback. Similarly, `fireOnOneErrback` will call `errback` as soon as any of the `Deferred`s call their `errback`. Note that `DeferredList` is still one-shot, like ordinary `Deferred`s, 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 `Deferred`s).

The `fireOnOneErrback` option is particularly useful when you want to wait for all the results if everything succeeds, but also want to know immediately if something fails.

The `consumeErrors` argument will stop the `DeferredList` from propagating any errors along the callback chains of any `Deferred`s it contains (usually creating a `DeferredList` has no effect on the results passed along the callbacks and `errbacks` of their `Deferred`s). Stopping errors at the `DeferredList` with this option will prevent “Unhandled error in `Deferred`” warnings from the `Deferred`s it contains without needing to add extra `errbacks`¹.

¹Unless of course a later callback starts a fresh error — but as we’ve already noted, adding callbacks to a `Deferred` after its used in a `DeferredList` is confusing and usually avoided.

3.4.5 Class Overview

This is an overview API reference for Deferred from the point of using a Deferred returned by a function. It is not meant to be a substitute for the docstrings in the Deferred class, but can provide guidelines for its use.

There is a parallel overview of functions used by the Deferred's *creator* in *Generating Deferreds* (page 109).

Basic Callback Functions

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

This is the method you will use to interact with Deferred. It adds a pair of callbacks “parallel” to each other (see diagram above) in the list of callbacks made when the Deferred is called back to. The signature of a method added using `addCallbacks` should be `myMethod(result, *methodArgs, **methodKeywords)`. If your method is passed in the callback slot, for example, all arguments in the tuple `callbackArgs` will be passed as `*methodArgs` to your method.

There are various convenience methods that are derivative of `addCallbacks`. I will not cover them in detail here, but it is important to know about them in order to create concise code.

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

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

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

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

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

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

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

Chaining Deferreds

If you need one Deferred to wait on another, all you need to do is return a Deferred from a method added to `addCallbacks`. Specifically, if you return Deferred B from a method added to Deferred A using `A.addCallbacks`, Deferred A's processing chain will stop until Deferred B's `.callback()` method is called; at that point, the next callback in A will be passed the result of the last callback in Deferred B's processing chain at the time.

If this seems confusing, don't worry about it right now – when you run into a situation where you need this behavior, you will probably recognize it immediately and realize why this happens. If you want to chain deferreds manually, there is also a convenience method to help you.

- `chainDeferred(otherDeferred)`

Add `otherDeferred` to the end of this Deferred's processing chain. When `self.callback` is called, the result of my processing chain up to this point will be passed to `otherDeferred.callback`. Further additions to my callback chain do not affect `otherDeferred`

This is the same as `self.addCallbacks(otherDeferred.callback, otherDeferred.errback)`

3.4.6 See also

1. *Generating Deferreds* (page 109), an introduction to writing asynchronous functions that return Deferreds.

3.5 Generating Deferreds

Deferred objects are signals that a function you have called does not yet have the data you want available. When a function returns a Deferred object, your calling function attaches callbacks to it to handle the data when available.

This document addresses the other half of the question: writing functions that return Deferreds, that is, constructing Deferred objects, arranging for them to be returned immediately without blocking until data is available, and firing their callbacks when the data is available.

This document assumes that you are familiar with the *asynchronous model* (page 8) used by Twisted, and with *using deferreds returned by functions* (page 99).

3.5.1 Class overview

This is an overview API reference for Deferred from the point of creating a Deferred and firing its callbacks and errbacks. It is not meant to be a substitute for the docstrings in the Deferred class, but can provide guidelines for its use.

There is a parallel overview of functions used by calling function which the Deferred is returned to at *Using Deferreds* (page 107).

Basic Callback Functions

- `callback(result)`

Run success callbacks with the given result. *This can only be run once.* Later calls to this or errback will raise `twisted.internet.defer.AlreadyCalledError`. If further callbacks or errbacks are added after this point, `addCallbacks` will run the callbacks immediately.

- `errback(failure)`

Run error callbacks with the given failure. *This can only be run once.* Later calls to this or callback will raise `twisted.internet.defer.AlreadyCalledError`. If further callbacks or errbacks are added after this point, `addCallbacks` will run the callbacks immediately.

3.5.2 What Deferreds don't do: make your code asynchronous

Deferreds do not make the code magically not block.

Let's take this function as an example:

```
from twisted.internet import defer

TARGET = 10000

def largeFibonacciNumber():
    # create a Deferred object to return:
    d = defer.Deferred()

    # calculate the ten thousandth Fibonacci number

    first = 0
    second = 1

    for i in xrange(TARGET - 1):
        new = first + second
        first = second
        second = new
        if i % 100 == 0:
            print "Progress: calculating the %dth Fibonacci number" % i

    # give the Deferred the answer to pass to the callbacks:
    d.callback(second)
```

```

    # return the Deferred with the answer:
    return d

import time

timeBefore = time.time()

# call the function and get our Deferred
d = largeFibonacciNumber()

timeAfter = time.time()

print "Total time taken for largeFibonacciNumber call: %0.3f seconds" % \
    (timeAfter - timeBefore)

# add a callback to it to print the number

def printNumber(number):
    print "The %dth Fibonacci number is %d" % (TARGET, number)

print "Adding the callback now."

d.addCallback(printNumber)

```

You will notice that despite creating a Deferred in the `largeFibonacciNumber` function, these things happened:

- the "Total time taken for largeFibonacciNumber call" output shows that the function did not return immediately as asynchronous functions are expected to do; and
- rather than the callback being added before the result was available and called after the result is available, it isn't even added until after the calculation has been completed.

The function completed its calculation before returning, blocking the process until it had finished, which is exactly what asynchronous functions are not meant to do. Deferreds are not a non-blocking talisman: they are a signal for asynchronous functions to *use* to pass results onto callbacks, but using them does not guarantee that you have an asynchronous function.

3.5.3 Advanced Processing Chain Control

- `pause()`
Cease calling any methods as they are added, and do not respond to callback, until `self.unpause()` is called.
- `unpause()`
If callback has been called on this Deferred already, call all the callbacks that have been added to this Deferred since `pause` was called.
Whether it was called or not, this will put this Deferred in a state where further calls to `addCallbacks` or `callback` will work as normal.

3.5.4 Returning Deferreds from synchronous functions

Sometimes you might wish to return a Deferred from a synchronous function. There are several reasons why, the major two are maintaining API compatibility with another version of your function which returns a Deferred, or allowing for the possibility that in the future your function might need to be asynchronous.

In the *Using Deferreds* (page 99) reference, we gave the following example of a synchronous function:

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

Source listing — *synch-validation.py*

While we can require that callers of our function wrap our synchronous result in a Deferred using maybe Deferred, for the sake of API compatibility it is better to return a Deferred ourselves using `defer.succeed`:

```
from twisted.internet import defer

def immediateIsValidUser(user):
    '''
    Returns a Deferred resulting in true if user is a valid user, false
    otherwise
    '''

    result = user in ["Alice", "Angus", "Agnes"]

    # return a Deferred object already called back with the value of result
    return defer.succeed(result)
```

There is an equivalent `defer.fail` method to return a Deferred with the errback chain already fired.

3.5.5 Integrating blocking code with Twisted

At some point, you are likely to need to call a blocking function: many functions in third party libraries will have long running blocking functions. There is no way to 'force' a function to be asynchronous: it must be written that way specifically. When using Twisted, your own code should be asynchronous, but there is no way to make third party functions asynchronous other than rewriting them.

In this case, Twisted provides the ability to run the blocking code in a separate thread rather than letting it block your application. The `twisted.internet.threads.deferToThread` function will set up a thread to run your blocking function, return a Deferred and later fire that Deferred when the thread completes.

Let's assume our `largeFibonacciNumber` function from above is in a third party library (returning the result of the calculation, not a Deferred) and is not easily modifiable to be finished in discrete blocks. This example shows it being called in a thread, unlike in the earlier section we'll see that the operation does not block our entire program:

```
def largeFibonacciNumber():
    """
    Represent a long running blocking function by calculating
    the TARGETth Fibonacci number
    """
    TARGET = 10000

    first = 0
    second = 1

    for i in xrange(TARGET - 1):
        new = first + second
        first = second
        second = new

    return second

from twisted.internet import threads, reactor
```



```

def fibonacciCallback(result):
    """
    Callback which manages the largeFibonacciNumber result by
    printing it out
    """
    print "largeFibonacciNumber result =", result
    # make sure the reactor stops after the callback chain finishes,
    # just so that this example terminates
    reactor.stop()

def run():
    """
    Run a series of operations, deferring the largeFibonacciNumber
    operation to a thread and performing some other operations after
    adding the callback
    """
    # get our Deferred which will be called with the largeFibonacciNumber result
    d = threads.deferToThread(largeFibonacciNumber)
    # add our callback to print it out
    d.addCallback(fibonacciCallback)
    # unless the largeFibonacciNumber thread returns very fast, these print
    # lines should happen first
    print "1st line after the addition of the callback"
    print "2nd line after the addition of the callback"

if __name__ == '__main__':
    run()
    reactor.run()

```

3.5.6 Possible sources of error

Deferreds greatly simplify the process of writing asynchronous code by providing a standard for registering callbacks, but there are some subtle and sometimes confusing rules that you need to follow if you are going to use them. This mostly applies to people who are writing new systems that use Deferreds internally, and not writers of applications that just add callbacks to Deferreds produced and processed by other systems. Nevertheless, it is good to know.

Firing Deferreds more than once is impossible

Deferreds are one-shot. You can only call `Deferred.callback` or `Deferred.errback` once. The processing chain continues each time you add new callbacks to an already-called-back-to-Deferred.

Synchronous callback execution

If a Deferred already has a result available, `addCallback` *may* call the callback synchronously: that is, immediately after it's been added. In situations where callbacks modify state, it might be desirable for the chain of processing to halt until all callbacks are added. For this, it is possible to pause and unpaue 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 unpaue it. The function adding the callbacks must unpaue a paused Deferred, it should *never* be the responsibility of the code that actually fires the callback chain by calling `callback` or `errback` as this would negate its usefulness!

3.6 Deferreds are beautiful! (A Tutorial)

3.6.1 Introduction

Deferreds are quite possibly the single most confusing topic that a newcomer to Twisted has to deal with. I am going to forgo the normal talk about what deferreds are, what they aren't, and why they're used in Twisted. Instead, I'm

going show you the logic behind what they *do*.

A deferred allows you to encapsulate the logic that you'd normally use to make a series of function calls after receiving a result into a single object. In the examples that follow, I'll first show you what's going to go on behind the scenes in the deferred chain, then show you the deferred API calls that set up that chain. All of these examples are runnable code, so feel free to play around with them.

3.6.2 A simple example

First, a simple example so that we have something to talk about:

```
#!/usr/bin/python2.3

from twisted.internet import defer
from twisted.python import failure, util

"""
here we have the simplest case, a single callback and a single errback
"""

num = 0

def handleFailure(f):
    print "errback"
    print "we got an exception: %s" % (f.getTraceback(),)
    f.trap(RuntimeError)

def handleResult(result):
    global num; num += 1
    print "callback %s" % (num,)
    print "\tgot result: %s" % (result,)
    return "yay! handleResult was successful!"

def behindTheScenes(result):
    # equivalent to d.callback(result)

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = handleResult(result)
        except:
            result = failure.Failure()
    else: # ---- errback
        pass

    if not isinstance(result, failure.Failure): # ---- callback
        pass
    else: # ---- errback
        try:
            result = handleFailure(result)
        except:
            result = failure.Failure()

def deferredExample():
    d = defer.Deferred()
    d.addCallback(handleResult)
    d.addErrback(handleFailure)
```

```

        d.callback("success")

if __name__ == '__main__':
    behindTheScenes("success")
    print "\n-----\n"
    global num; num = 0
    deferredExample()

```

Source listing — *deferred_ex.py*

And the output: (since both methods in the example produce the same output, it will only be shown once.)

```

callback 1
    got result: success

```

Here we have the simplest case. A deferred with a single callback and a single errback. Normally, a function would create a deferred and hand it back to you when you request an operation that needs to wait for an event for completion. The object you called then does `d.callback(result)` when the results are in.

The thing to notice is that there is only one result that is passed from method to method, and that the result returned from a method is the argument to the next method in the chain. In case of an exception, result is set to an instance of `Failure` that describes the exception.

3.6.3 Errbacks

Failure in requested operation

Things don't always go as planned, and sometimes the function that returned the deferred needs to alert the callback chain that an error has occurred.

```

#!/usr/bin/python2.3

from twisted.internet import defer
from twisted.python import failure, util

"""
this example is analogous to a function calling .errback(failure)
"""

class Counter(object):
    num = 0

def handleFailure(f):
    print "errback"
    print "we got an exception: %s" % (f.getTraceback(),)
    f.trap(RuntimeError)

def handleResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    return "yay! handleResult was successful!"

def failAtHandlingResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)

```

```

    print "\tgot result: %s" % (result,)
    print "\tabout to raise exception"
    raise RuntimeError, "whoops! we encountered an error"

def behindTheScenes(result):
    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = handleResult(result)
        except:
            result = failure.Failure()
    else: # ---- errback
        pass

    if not isinstance(result, failure.Failure): # ---- callback
        pass
    else: # ---- errback
        try:
            result = handleFailure(result)
        except:
            result = failure.Failure()

def deferredExample(result):
    d = defer.Deferred()
    d.addCallback(handleResult)
    d.addCallback(failAtHandlingResult)
    d.addErrback(handleFailure)

    d.errback(result)

if __name__ == '__main__':
    result = None
    try:
        raise RuntimeError, "*doh*! failure!"
    except:
        result = failure.Failure()
    behindTheScenes(result)
    print "\n-----\n"
    Counter.num = 0
    deferredExample(result)

```

Source listing — *deferred_ex1a.py*

```

errback
we got an exception: Traceback (most recent call last):
--- exception caught here ---
  File "deferred_ex1a.py", line 73, in ?
    raise RuntimeError, "*doh*! failure!"
exceptions.RuntimeError: *doh*! failure!

```

The important thing to note (as it will come up again in later examples) is that the callback isn't touched, the failure goes right to the errback. Also note that the errback trap(s) the expected exception type. If you don't trap the exception, an error will be logged when the deferred is garbage-collected.

Exceptions raised in callbacks

Now let's see what happens when *our callback* raises an exception

```
#!/usr/bin/python2.3

from twisted.internet import defer
from twisted.python import failure, util

"""
here we have a slightly more involved case. The deferred is called back with a
result. the first callback returns a value, the second callback, however
raises an exception, which is handled by the errback.
"""

class Counter(object):
    num = 0

def handleFailure(f):
    print "errback"
    print "we got an exception: %s" % (f.getTraceback(),)
    f.trap(RuntimeError)

def handleResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    return "yay! handleResult was successful!"

def failAtHandlingResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    print "\tabout to raise exception"
    raise RuntimeError, "whoops! we encountered an error"

def behindTheScenes(result):
    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = handleResult(result)
        except:
            result = failure.Failure()
    else: # ---- errback
        pass

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = failAtHandlingResult(result)
        except:
            result = failure.Failure()
    else: # ---- errback
        pass

    if not isinstance(result, failure.Failure): # ---- callback
```

```

        pass
    else:
        # ---- errback
        try:
            result = handleFailure(result)
        except:
            result = failure.Failure()

def deferredExample():
    d = defer.Deferred()
    d.addCallback(handleResult)
    d.addCallback(failAtHandlingResult)
    d.addErrback(handleFailure)

    d.callback("success")

if __name__ == '__main__':
    behindTheScenes("success")
    print "\n-----\n"
    Counter.num = 0
    deferredExample()

```

Source listing — *deferred_ex1b.py*

And the output: (note, tracebacks will be edited slightly to conserve space)

```

callback 1
    got result: success
callback 2
    got result: yay! handleResult was successful!
    about to raise exception
errback
we got an exception: Traceback (most recent call last):
--- <exception caught here> ---
  File "/home/slyphon/Projects/Twisted/trunk/twisted/internet/defer.py", line
326, in _runCallbacks
    self.result = callback(self.result, *args, **kw)
  File "./deferred_ex1.py", line 32, in failAtHandlingResult
    raise RuntimeError, "whoops! we encountered an error"
exceptions.RuntimeError: whoops! we encountered an error

```

If your callback raises an exception, the next method to be called will be the next errback in your chain.

Exceptions will only be handled by errbacks

If a callback raises an exception the next method to be called will be next errback in the chain. If the chain is started off with a failure, the first method to be called will be the first errback.

```
#!/usr/bin/python2.3
```

```

from twisted.internet import defer
from twisted.python import failure, util

```

```
"""
```

```

this example shows an important concept that many deferred newbies
(myself included) have trouble understanding.

```

when an error occurs in a callback, the first errback after the error occurs will be the next method called. (in the next example we'll see what happens in the 'chain' after an errback)

```
"""

class Counter(object):
    num = 0

def handleFailure(f):
    print "errback"
    print "we got an exception: %s" % (f.getTraceback(),)
    f.trap(RuntimeError)

def handleResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    return "yay! handleResult was successful!"

def failAtHandlingResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    print "\tabout to raise exception"
    raise RuntimeError, "whoops! we encountered an error"

def behindTheScenes(result):
    # equivalent to d.callback(result)

    # now, let's make the error happen in the first callback

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = failAtHandlingResult(result)
        except:
            result = failure.Failure()
    else:
        # ---- errback
        pass

    # note: this callback will be skipped because
    # result is a failure

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = handleResult(result)
        except:
            result = failure.Failure()
    else:
        # ---- errback
        pass

    if not isinstance(result, failure.Failure): # ---- callback
        pass
```

```

        else:
            # ---- errback
            try:
                result = handleFailure(result)
            except:
                result = failure.Failure()

def deferredExample():
    d = defer.Deferred()
    d.addCallback(failAtHandlingResult)
    d.addCallback(handleResult)
    d.addErrback(handleFailure)

    d.callback("success")

if __name__ == '__main__':
    behindTheScenes("success")
    print "\n-----\n"
    Counter.num = 0
    deferredExample()

```

Source listing — *deferred_ex2.py*

```

callback 1
    got result: success
    about to raise exception
errback
we got an exception: Traceback (most recent call last):
  File "./deferred_ex2.py", line 85, in ?
    nonDeferredExample("success")
--- <exception caught here> ---
  File "./deferred_ex2.py", line 46, in nonDeferredExample
    result = failAtHandlingResult(result)
  File "./deferred_ex2.py", line 35, in failAtHandlingResult
    raise RuntimeError, "whoops! we encountered an error"
exceptions.RuntimeError: whoops! we encountered an error

```

You can see that our second callback, `handleResult` was not called because `failAtHandlingResult` raised an exception

Handling an exception and continuing on

In this example, we see an errback handle an exception raised in the preceding callback. Take note that it could just as easily been an exception from *any other* preceding method. You'll see that after the exception is handled in the errback (i.e. the errback does not return a failure or raise an exception) the chain continues on with the next callback.

```
#!/usr/bin/python2.3
```

```

from twisted.internet import defer
from twisted.python import failure, util

"""
now we see how an errback can handle errors. if an errback
does not raise an exception, the next callback in the chain
will be called

```



```

"""

class Counter(object):
    num = 0

def handleFailure(f):
    print "errback"
    print "we got an exception: %s" % (f.getTraceback(),)
    f.trap(RuntimeError)
    return "okay, continue on"

def handleResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    return "yay! handleResult was successful!"

def failAtHandlingResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    print "\tabout to raise exception"
    raise RuntimeError, "whoops! we encountered an error"

def callbackAfterErrback(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)

def behindTheScenes(result):
    # equivalent to d.callback(result)

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = handleResult(result)
        except:
            result = failure.Failure()
    else: # ---- errback
        pass

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = failAtHandlingResult(result)
        except:
            result = failure.Failure()
    else: # ---- errback
        pass

    if not isinstance(result, failure.Failure): # ---- callback
        pass
    else: # ---- errback

```

```

        try:
            result = handleFailure(result)
        except:
            result = failure.Failure()

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = callbackAfterErrback(result)
        except:
            result = failure.Failure()
    else: # ---- errback
        pass

def deferredExample():
    d = defer.Deferred()
    d.addCallback(handleResult)
    d.addCallback(failAtHandlingResult)
    d.addErrback(handleFailure)
    d.addCallback(callbackAfterErrback)

    d.callback("success")

if __name__ == '__main__':
    behindTheScenes("success")
    print "\n-----\n"
    Counter.num = 0
    deferredExample()

```

Source listing — *deferred_ex3.py*

```

callback 1
    got result: success
    about to raise exception
errback
we got an exception: Traceback (most recent call last):
--- <exception caught here> ---
  File "/home/slyphon/Projects/Twisted/trunk/twisted/internet/defer.py", line
326, in _runCallbacks
    self.result = callback(self.result, *args, **kw)
  File "./deferred_ex2.py", line 35, in failAtHandlingResult
    raise RuntimeError, "whoops! we encountered an error"
exceptions.RuntimeError: whoops! we encountered an error

```

3.6.4 addBoth: the deferred version of finally

Now we see how deferreds do *finally*, with `.addBoth`. The callback that gets added as `addBoth` will be called if the result is a failure or non-failure. We'll also see in this example, that our `doThisNoMatterWhat()` method follows a common idiom in deferred callbacks by acting as a passthru, returning the value that it received to allow processing the chain to continue, but appearing transparent in terms of the result.

```
#!/usr/bin/python2.3
```

```
from twisted.internet import defer
```

```

from twisted.python import failure, util

"""
now we'll see what happens when you use 'addBoth'
"""

class Counter(object):
    num = 0

def handleFailure(f):
    print "errback"
    print "we got an exception: %s" % (f.getTraceback(),)
    f.trap(RuntimeError)
    return "okay, continue on"

def handleResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    return "yay! handleResult was successful!"

def failAtHandlingResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    print "\tabout to raise exception"
    raise RuntimeError, "whoops! we encountered an error"

def doThisNoMatterWhat(arg):
    Counter.num += 1
    print "both %s" % (Counter.num,)
    print "\tgot argument %r" % (arg,)
    print "\tdoing something very important"
    # we pass the argument we received to the next phase here
    return arg

def behindTheScenes(result):
    # equivalent to d.callback(result)

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = handleResult(result)
        except:
            result = failure.Failure()
    else: # ---- errback
        pass

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = failAtHandlingResult(result)
        except:
            result = failure.Failure()

```

```

    else:
        pass

    # ---- errback

    # ---- this is equivalent to addBoth(doThisNoMatterWhat)

    if not isinstance(result, failure.Failure):
        try:
            result = doThisNoMatterWhat(result)
        except:
            result = failure.Failure()
    else:
        try:
            result = doThisNoMatterWhat(result)
        except:
            result = failure.Failure()

    if not isinstance(result, failure.Failure): # ---- callback
        pass
    else:
        # ---- errback
        try:
            result = handleFailure(result)
        except:
            result = failure.Failure()

def deferredExample():
    d = defer.Deferred()
    d.addCallback(handleResult)
    d.addCallback(failAtHandlingResult)
    d.addBoth(doThisNoMatterWhat)
    d.addErrback(handleFailure)

    d.callback("success")

if __name__ == '__main__':
    behindTheScenes("success")
    print "\n-----\n"
    Counter.num = 0
    deferredExample()

```

Source listing — *deferred_ex4.py*

```

callback 1
    got result: success
callback 2
    got result: yay! handleResult was successful!
    about to raise exception
both 3
    got argument <twisted.python.failure.Failure exceptions.RuntimeError>
    doing something very important
errback
we got an exception: Traceback (most recent call last):
--- <exception caught here> ---

```

```

File "/home/slyphon/Projects/Twisted/trunk/twisted/internet/defer.py", line
326, in _runCallbacks
    self.result = callback(self.result, *args, **kw)
File "./deferred_ex4.py", line 32, in failAtHandlingResult
    raise RuntimeError, "whoops! we encountered an error"
exceptions.RuntimeError: whoops! we encountered an error

```

You can see that the errback is called, (and consequently, the failure is trapped). This is because `doThisNoMatter-What` method returned the value it received, a failure.

3.6.5 addCallbacks: decision making based on previous success or failure

As we've been seeing in the examples, the callback is a pair of callback/errback. Using `addCallback` or `addErrback` is actually a special case where one of the pair is a pass statement. If you want to make a decision based on whether or not the previous result in the chain was a failure or not (which is very rare, but included here for completeness), you use `addCallbacks`. Note that this is *not* the same thing as an `addCallback` followed by an `addErrback`.

```

#!/usr/bin/python2.3

from twisted.internet import defer
from twisted.python import failure, util

"""
now comes the more nuanced addCallbacks, which allows us to make a
yes/no (branching) decision based on whether the result at a given point is
a failure or not.
"""

class Counter(object):
    num = 0

def handleFailure(f):
    print "errback"
    print "we got an exception: %s" % (f.getTraceback(),)
    f.trap(RuntimeError)
    return "okay, continue on"

def handleResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    return "yay! handleResult was successful!"

def failAtHandlingResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    print "\tabout to raise exception"
    raise RuntimeError, "whoops! we encountered an error"

def yesDecision(result):
    Counter.num += 1
    print "yes decision %s" % (Counter.num,)
    print "\twasn't a failure, so we can plow ahead"
    return "go ahead!"

```

```

def noDecision(result):
    Counter.num += 1
    result.trap(RuntimeError)
    print "no decision %s" % (Counter.num,)
    print "\t*doh*! a failure! quick! damage control!"
    return "damage control successful!"

def behindTheScenes(result):

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = failAtHandlingResult(result)
        except:
            result = failure.Failure()
    else: # ---- errback
        pass

    # this is equivalent to addCallbacks(yesDecision, noDecision)

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = yesDecision(result)
        except:
            result = failure.Failure()
    else: # ---- errback
        try:
            result = noDecision(result)
        except:
            result = failure.Failure()

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = handleResult(result)
        except:
            result = failure.Failure()
    else: # ---- errback
        pass

    # this is equivalent to addCallbacks(yesDecision, noDecision)

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = yesDecision(result)
        except:
            result = failure.Failure()
    else: # ---- errback
        try:
            result = noDecision(result)
        except:
            result = failure.Failure()

```

```

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = handleResult(result)
        except:
            result = failure.Failure()
    else: # ---- errback
        pass

    if not isinstance(result, failure.Failure): # ---- callback
        pass
    else: # ---- errback
        try:
            result = handleFailure(result)
        except:
            result = failure.Failure()

def deferredExample():
    d = defer.Deferred()
    d.addCallback(failAtHandlingResult)
    d.addCallbacks(yesDecision, noDecision) # noDecision will be called
    d.addCallback(handleResult) # - A -
    d.addCallbacks(yesDecision, noDecision) # yesDecision will be called
    d.addCallback(handleResult)
    d.addErrback(handleFailure)

    d.callback("success")

if __name__ == '__main__':
    behindTheScenes("success")
    print "\n-----\n"
    Counter.num = 0
    deferredExample()

```

Source listing — *deferred_ex5.py*

```

callback 1
    got result: success
    about to raise exception
no decision 2
    *doh*! a failure! quick! damage control!
callback 3
    got result: damage control successful!
yes decision 4
    wasn't a failure, so we can plow ahead
callback 5
    got result: go ahead!

```

Notice that our errback is never called. The `noDecision` method returns a non-failure so processing continues with the next callback. If we wanted to skip the callback at "- A -" because of the error, but do some kind of processing in response to the error, we would have used a passthru, and returned the failure we received, as we see in this next example:

```
#!/usr/bin/python2.3
```

```

from twisted.internet import defer
from twisted.python import failure, util

"""
now comes the more nuanced addCallbacks, which allows us to make a
yes/no (branching) decision based on whether the result at a given point is
a failure or not.

here, we return the failure from noDecisionPassthru, the errback argument to
the first addCallbacks method invocation, and see what happens

"""

class Counter(object):
    num = 0

def handleFailure(f):
    print "errback"
    print "we got an exception: %s" % (f.getTraceback(),)
    f.trap(RuntimeError)
    return "okay, continue on"

def handleResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    return "yay! handleResult was successful!"

def failAtHandlingResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    print "\tabout to raise exception"
    raise RuntimeError, "whoops! we encountered an error"

def yesDecision(result):
    Counter.num += 1
    print "yes decision %s" % (Counter.num,)
    print "\twasn't a failure, so we can plow ahead"
    return "go ahead!"

def noDecision(result):
    Counter.num += 1
    result.trap(RuntimeError)
    print "no decision %s" % (Counter.num,)
    print "\tdoh*! a failure! quick! damage control!"
    return "damage control successful!"

def noDecisionPassthru(result):
    Counter.num += 1
    print "no decision %s" % (Counter.num,)
    print "\tdoh*! a failure! don't know what to do, returning failure!"
    return result

def behindTheScenes(result):

```



```

if not isinstance(result, failure.Failure): # ---- callback
    try:
        result = failAtHandlingResult(result)
    except:
        result = failure.Failure()
else:
    pass # ---- errback

# this is equivalent to addCallbacks(yesDecision, noDecision)

if not isinstance(result, failure.Failure): # ---- callback
    try:
        result = yesDecision(result)
    except:
        result = failure.Failure()
else:
    pass # ---- errback
    try:
        result = noDecisionPassthru(result)
    except:
        result = failure.Failure()

if not isinstance(result, failure.Failure): # ---- callback
    try:
        result = handleResult(result)
    except:
        result = failure.Failure()
else:
    pass # ---- errback

# this is equivalent to addCallbacks(yesDecision, noDecision)

if not isinstance(result, failure.Failure): # ---- callback
    try:
        result = yesDecision(result)
    except:
        result = failure.Failure()
else:
    pass # ---- errback
    try:
        result = noDecision(result)
    except:
        result = failure.Failure()

if not isinstance(result, failure.Failure): # ---- callback
    try:
        result = handleResult(result)
    except:
        result = failure.Failure()
else:
    pass # ---- errback

if not isinstance(result, failure.Failure): # ---- callback

```

```

        pass
    else:
        # ---- errback
        try:
            result = handleFailure(result)
        except:
            result = failure.Failure()

def deferredExample():
    d = defer.Deferred()
    d.addCallback(failAtHandlingResult)

    # noDecisionPassthru will be called
    d.addCallbacks(yesDecision, noDecisionPassthru)
    d.addCallback(handleResult) # - A -

    # noDecision will be called
    d.addCallbacks(yesDecision, noDecision)
    d.addCallback(handleResult) # - B -
    d.addErrback(handleFailure)

    d.callback("success")

if __name__ == '__main__':
    behindTheScenes("success")
    print "\n-----\n"
    Counter.num = 0
    deferredExample()

```

Source listing — *deferred_ex6.py*

```

callback 1
    got result: success
    about to raise exception
no decision 2
    *doh*! a failure! don't know what to do, returning failure!
no decision 3
    *doh*! a failure! quick! damage control!
callback 4
    got result: damage control successful!

```

Two things to note here. First, "- A -" was skipped, like we wanted it to, and the second thing is that after "- A -", noDecision is called, because *it is the next errback that exists in the chain*. It returns a non-failure, so processing continues with the next callback at "- B -", and the errback at the end of the chain is never called

3.6.6 Hints, tips, common mistakes, and miscellaney

The deferred callback chain is stateful

A deferred that has been called back will call its addCallback and addErrback methods as appropriate in the order they are added, when they are added. So we see in the following example, deferredExample1 and deferredExample2 are equivalent. The first sets up the processing chain beforehand and then executes it, the other executes the chain as it is being constructed. This is because deferreds are *stateful*.

```

#!/usr/bin/python2.3

from twisted.internet import defer

```

```

from twisted.python import failure, util

"""
The deferred callback chain is stateful, and can be executed before
or after all callbacks have been added to the chain
"""

class Counter(object):
    num = 0

def handleFailure(f):
    print "errback"
    print "we got an exception: %s" % (f.getTraceback(),)
    f.trap(RuntimeError)

def handleResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    return "yay! handleResult was successful!"

def failAtHandlingResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    print "\tabout to raise exception"
    raise RuntimeError, "whoops! we encountered an error"

def deferredExample1():
    # this is another common idiom, since all add* methods
    # return the deferred instance, you can just chain your
    # calls to addCallback and addErrback

    d = defer.Deferred().addCallback(failAtHandlingResult
                                    ).addCallback(handleResult
                                    ).addErrback(handleFailure)

    d.callback("success")

def deferredExample2():
    d = defer.Deferred()

    d.callback("success")

    d.addCallback(failAtHandlingResult)
    d.addCallback(handleResult)
    d.addErrback(handleFailure)

if __name__ == '__main__':
    deferredExample1()
    print "\n-----\n"
    Counter.num = 0
    deferredExample2()

```

```

callback 1
    got result: success
    about to raise exception
errback
we got an exception: Traceback (most recent call last):
--- <exception caught here> ---
  File "/home/slyphon/Projects/Twisted/trunk/twisted/internet/defer.py", line
326, in _runCallbacks
    self.result = callback(self.result, *args, **kw)
  File "./deferred_ex7.py", line 35, in failAtHandlingResult
    raise RuntimeError, "whoops! we encountered an error"
exceptions.RuntimeError: whoops! we encountered an error

```

```

-----

callback 1
    got result: success
    about to raise exception
errback
we got an exception: Traceback (most recent call last):
--- <exception caught here> ---
  File "/home/slyphon/Projects/Twisted/trunk/twisted/internet/defer.py", line
326, in _runCallbacks
    self.result = callback(self.result, *args, **kw)
  File "./deferred_ex7.py", line 35, in failAtHandlingResult
    raise RuntimeError, "whoops! we encountered an error"
exceptions.RuntimeError: whoops! we encountered an error

```

This example also shows you the common idiom of chaining calls to `addCallback` and `addErrback`.

Don't call `.callback()` on deferreds you didn't create!

It is an error to reinvoke deferreds `callback` or `errback` method, therefore if you didn't create a deferred, *do not under any circumstances* call its `callback` or `errback`. doing so will raise an exception

Callbacks can return deferreds

If you need to call a method that returns a deferred within your callback chain, just return that deferred, and the result of the secondary deferred's processing chain will become the result that gets passed to the next callback of the primary deferreds processing chain

```

#!/usr/bin/python2.3

from twisted.internet import defer
from twisted.python import failure, util

"""
"""

class Counter(object):
    num = 0
    let = 'a'

    def incrLet(cls):
        cls.let = chr(ord(cls.let) + 1)
    incrLet = classmethod(incrLet)

```

```

def handleFailure(f):
    print "errback"
    print "we got an exception: %s" % (f.getTraceback(),)
    return f

def subCb_B(result):
    print "sub-callback %s" % (Counter.let,)
    Counter.incrLet()
    s = " beautiful!"
    print "\tadding %r to result" % (s,)
    result += s
    return result

def subCb_A(result):
    print "sub-callback %s" % (Counter.let,)
    Counter.incrLet()
    s = " are "
    print "\tadding %r to result" % (s,)
    result += s
    return result

def mainCb_1(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    result += " Deferreds "

    d = defer.Deferred().addCallback(subCb_A
                                    ).addCallback(subCb_B)
    d.callback(result)
    return d

def mainCb_2(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)

def deferredExample():
    d = defer.Deferred().addCallback(mainCb_1
                                    ).addCallback(mainCb_2)

    d.callback("I hope you'll agree: ")

if __name__ == '__main__':
    deferredExample()

```

Source listing — *deferred_ex8.py*

```

callback 1
    got result: I hope you'll agree:
sub-callback a
    adding ' are ' to result
sub-callback b

```

```

        adding ' beautiful!' to result
callback 2
        got result: I hope you'll agree:  Deferreds  are  beautiful!

```

3.6.7 Conclusion

Deferreds can be confusing, but only because they're so elegant and simple. There is a lot of logical power that can be expressed with a deferred's processing chain, and once you see what's going on behind the curtain, it's a lot easier to understand how to make use of what deferreds have to offer.

3.7 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 3.5 seconds after it was scheduled: %s" % s

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

# f() will only be called if the event loop was started:
reactor.run()

```

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

```

from twisted.internet import task

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

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

# l.stop() will stop the looping calls
reactor.run()

```

If we want to cancel a task that we've scheduled:

```

from twisted.internet import reactor

def f():
    print "I'll never run."

callID = reactor.callLater(5, f)
callID.cancel()
reactor.run()

```

As with all reactor-based code, in order for scheduling to work the reactor must be started using `reactor.run()`.

3.8 Using Threads in Twisted

3.8.1 Running code in a thread-safe manner

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

```

from twisted.internet import reactor

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

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

```

3.8.2 Running code in threads

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

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

```

from twisted.internet import reactor

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

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

```

3.8.3 Utility Methods

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

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

```

from twisted.internet import threads

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

def aSillyBlockingMethodTwo(x):
    print x

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

```

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

```

from twisted.internet import threads

def doLongCalculation():
    # .... do long calculation here ...
    return 3

def printResult(x):

```

```

print x

# run method in thread and get result as defer.Deferred
d = threads.deferToThread(doLongCalculation)
d.addCallback(printResult)

```

3.8.4 Managing the Thread Pool

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

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

```

from twisted.internet import reactor

reactor.suggestThreadPoolSize(30)

```

The default size of the thread pool depends on the reactor being used; the default reactor uses a minimum size of 5 and a maximum size of 10. Be careful that you understand threads and their resource usage before drastically altering the thread pool sizes.

3.9 Choosing a Reactor and GUI Toolkit Integration

3.9.1 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 136)
- *The poll()-based reactor* (page 136)

Platform-specific reactor implementations exist for:

- *KQueue for FreeBSD and OS X* (page 136)
- *Win32 (WFMO)* (page 136)
- *Win32 (IOCP)* (page 136)
- *Mac OS X* (page 137)

The remaining custom reactor implementations provide support for integrating with the native event loops of various graphical toolkits. This lets your Twisted application use all of the usual Twisted APIs while still being a graphical application.

Twisted currently integrates with the following graphical toolkits:

- *GTK+ 1.2 and 2.0* (page 137)
- *Tkinter* (page 137)
- *WxPython* (page 137)
- *Win32* (page 136)
- *CoreFoundation* (page 137)
- *PyUI* (page 138)

When using applications that runnable using `twistd`, e.g. TAPs or plugins, there is no need to choose a reactor explicitly, since this can be chosen using `twistd`'s `-r` option.

In all cases, the event loop is started by calling `reactor.run()`. In all cases, the event loop should be stopped with `reactor.stop()`.

IMPORTANT: installing a reactor should be the first thing done in the app, since any code that does from `twisted.internet import reactor` will automatically install the default reactor if the code hasn't already installed one.

3.9.2 Reactor Functionality

	Status	TCP	SSL	UDP	Threading	Processes	Scheduling	Platforms
select()	Stable	Y	Y	Y	Y	Y	Y	Unix, Win32
poll()	Stable	Y	Y	Y	Y	Y	Y	Unix
Win32 (WFMO)	Experimental	Y	Y	Y	Y	Y	Y	Win32
Win32 (IOCP)	Experimental	Y	N	N	N	N	Y	Win32
CoreFoundation	Unmaintained	Y	Y	Y	Y	Y	Y	OS X
Gtk	Stable	Y	Y	Y	Y	Y	Y	Unix, Win32
wx	Experimental	Y	Y	Y	Y	Y	Y	Unix, Win32
kqueue	Experimental	Y	Y	Y	Y	Y	Y	FreeBSD

Table 3.1: Summary of reactor features

3.9.3 General Purpose Reactors

Select()-based Reactor

The select reactor is currently the default reactor on all platforms. The following code will install it, if no other reactor has been installed:

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

In the future, if another reactor becomes the default, but the select reactor is desired, it may be installed via:

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

Poll()-based Reactor

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

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

3.9.4 Platform-Specific Reactors

KQueue

The KQueue Reactor allows Twisted to use FreeBSD's kqueue mechanism for event scheduling. See instructions in the `twisted.internet.kqreactor`'s docstring for installation notes.

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

Win32 (WFMO)

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()
```

Win32 (IOCP)

Windows provides a fast, scalable event notification system known as IO Completion Ports, or IOCP for short. An extremely experimental reactor based on IOCP is provided with Twisted.

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

3.9.5 GUI Integration Reactors

GTK+

Twisted integrates with PyGTK², versions 1.2 (`gtkreactor`) and 2.0 (`gtk2reactor`). Sample applications using GTK+ and Twisted are available in the Twisted SVN.

GTK-2.0 split the event loop out of the GUI toolkit, into a separate module called “glib”. To run an application using the glib event loop, use the `glib2reactor`. This will be slightly faster than `gtk2reactor` (and does not require a working X display), but cannot be used to run GUI applications.

```
from twisted.internet import gtkreactor # for gtk-1.2
gtkreactor.install()

from twisted.internet import gtk2reactor # for gtk-2.0
gtk2reactor.install()

from twisted.internet import glib2reactor # for non-GUI apps
glib2reactor.install()
```

CoreFoundation

Twisted integrates with PyObjC³, version 1.0. Sample applications using Cocoa and Twisted are available in the examples directory under Cocoa.

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

3.9.6 Non-Reactor GUI Integration

Tkinter

The support for Tkinter⁴ doesn’t use a specialized reactor. Instead, there is some specialized support code:

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

root = Tk()

# Install the Reactor support
tksupport.install(root)

# at this point build Tk app as usual using the root object,
# and start the program with "reactor.run()", and stop it
# with "reactor.stop()".
```

wxPython

Twisted currently supports two methods of integrating wxPython. Unfortunately, neither method will work on all wxPython platforms (such as GTK2 or Windows). It seems that the only portable way to integrate with wxPython is to run it in a separate thread. One of these methods may be sufficient if your wx app is limited to a single platform.

As with *Tkinter* (this page), the support for integrating Twisted with a wxPython⁵ 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)
```

²<http://www.daa.com.au/~james/pygtk/>

³<http://pyobjc.sf.net/>

⁴<http://www.python.org/topics/tkinter/>

⁵<http://www.wxpython.org>

However, this has issues when running on Windows, so Twisted now comes with alternative wxPython support using a reactor. Using this method is probably better. Initialization is done in two stages. In the first, the reactor is installed:

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

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

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

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

PyUI

As with *Tkinter* (page 137), the support for integrating Twisted with a PyUI⁶ application uses specialized support code rather than a simple reactor.

```
from twisted.internet import pyuisupport, reactor

pyuisupport.install(args=(640, 480), kw={'renderer': 'gl'})
```

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

⁶<http://pyui.sourceforge.net>

Chapter 4

High-Level Twisted

4.1 The Basics

4.1.1 Application

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

Other HOWTOs describe how to write custom code for Applications, but this one describes how to use already written code (which can be part of Twisted or from a third-party Twisted plugin developer). The Twisted distribution comes with an important tool to deal with Applications, `twistd`.

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

4.1.2 `twistd`

The Twisted Daemon is a program that knows how to run Applications. This program is `twistd(1)`. Strictly speaking, `twistd` is not necessary – fetching the application, getting the `IService` component, calling `startService`, scheduling `stopService` when the reactor shuts down, and then calling `reactor.run()` could be done manually. `twistd(1)`, however, supplies many options which are highly useful for program set up.

`twistd` supports choosing a reactor (for more on reactors, see *Choosing a Reactor* (page 135)), logging to a logfile, daemonizing and more. `twistd` supports all Applications mentioned above – and an additional one. Sometimes it is convenient to write the code for building a class in straight Python. One big source of such Python files is the `doc/examples` directory. When a straight Python file which defines an Application object called `application` is used, use the `-y` option.

When `twistd` runs, it records its process id in a `twistd.pid` file (this can be configured via a command line switch). In order to shutdown the `twistd` process, kill that pid (usually you would do `kill `cat twistd.pid``).

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

4.1.3 `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 archive¹.

4.1.4 tap2rpm

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

4.2 The Twisted Plugin System

The purpose of this guide is to describe the preferred way to write extensible Twisted applications (and consequently, also to describe how to extend applications written in such a way). This extensibility is achieved through the definition of one or more APIs and a mechanism for collecting code plugins which implement this API to provide some additional functionality. At the base of this system is the `twisted.plugin` module.

Making an application extensible using the plugin system has several strong advantages over other techniques:

- It allows third-party developers to easily enhance your software in a way that is loosely coupled: only the plugin API is required to remain stable.
- It allows new plugins to be discovered flexibly. For example, plugins can be loaded and saved when a program is first run, or re-discovered each time the program starts up, or they can be polled for repeatedly at runtime (allowing the discovery of new plugins installed after the program has started).

4.2.1 Writing Extensible Programs

Taking advantage of `twisted.plugin` is a two step process:

1. Define an interface which plugins will be required to implement. This is done using the `zope.interface` package in the same way one would define an interface for any other purpose.

A convention for defining interfaces is to do so in a file named like *ProjectName/projectname/projectname.py*. The rest of this document will follow that convention: consider the following interface definition to be in `Matsim/matsim/imatsim.py`, an interface definition module for a hypothetical material simulation package.

2. At one or more places in your program, invoke `twisted.plugin.getPlugins` and iterate over its result.

As an example of the first step, consider the following interface definition for a physical modelling system.

```
from zope.interface import Interface, Attribute

class IMaterial(Interface):
    """
    An object with specific physical properties
    """
    def yieldStress(temperature):
        """
        Returns the pressure this material can support without
        fracturing at the given temperature.

        @type temperature: C{float}
        @param temperature: Kelvins

        @rtype: C{float}
        @return: Pascals
        """

    dielectricConstant = Attribute(" ")
```

¹<http://twistedmatrix.com/users/moshez/apt>

```
@type dielectricConstant: C{complex}
@ivar dielectricConstant: The relative permittivity, with the
real part giving reflective surface properties and the
imaginary part giving the radio absorption coefficient.
""" )
```

In another module, we might have a function that operates on objects providing the `IMaterial` interface:

```
def displayMaterial(m):
    print 'A material with yield stress %s at 500 K' % (m.yieldStress(500),)
    print 'Also a dielectric constant of %s.' % (m.dielectricConstant,)
```

The last piece of required code is that which collects `IMaterial` providers and passes them to the `displayMaterial` function.

```
from twisted.plugin import getPlugins
from matsim import imatsim

def displayAllKnownMaterials():
    for material in getPlugins(imatsim.IMaterial):
        displayMaterial(material)
```

Third party developers may now contribute different materials to be used by this modelling system by implementing one or more plugins for the `IMaterial` interface.

4.2.2 Extending an Existing Program

The above code demonstrates how an extensible program might be written using Twisted's plugin system. How do we write plugins for it, though? Essentially, we create objects which provide the required interface and then make them available at a particular location. Consider the following example.

```
from twisted.plugin import IPlugin
from matsim import imatsim

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

    def __init__(self, yieldStressFactor, dielectricConstant):
        self._yieldStressFactor = yieldStressFactor
        self.dielectricConstant = dielectricConstant

    def yieldStress(self, temperature):
        return self._yieldStressFactor * temperature

steelPlate = SimpleMaterial(2.06842719e11, 2.7 + 0.2j)
brassPlate = SimpleMaterial(1.03421359e11, 1.4 + 0.5j)
```

`steelPlate` and `brassPlate` now provide both `IPlugin` and `IMaterial`. All that remains is to make this module available at an appropriate location. For this, there are two options. The first of these is primarily useful during development: if a directory which has been added to `sys.path` (typically by adding it to the `PYTHONPATH` environment variable) contains a *directory* (not a Python package) named `twisted/plugins/`, each `.py` file in that directory will be loaded as a source of plugins. Second, each module in the installed version of Twisted's `twisted.plugins` package will also be loaded as a source of plugins.

Once this plugin is installed in one of these two ways, `displayAllKnownMaterials` can be run and we will see two pairs of output: one for a steel plate and one for a brass plate.

4.2.3 Alternate Plugin Packages

`getPlugins` takes one additional argument not mentioned above. If passed in, the 2nd argument should be a module or package to be used instead of `twisted.plugins` as the plugin meta-package. If you are writing a plugin for a

Twisted interface, you should never need to pass this argument. However, if you have developed an interface of your own, you may want to mandate that plugins for it are installed in your own plugins package, rather than in Twisted's. In this case, you probably also want to support `yourproject/plugins/` directories for ease of development. To do so, you should make the `__init__.py` for that package contain at least the following lines.

```
import os, sys
__path__ = [os.path.abspath(os.path.join(x, 'yourproject', 'plugins'))
             for x in sys.path]

__all__ = []
```

The key behavior here is that interfaces are essentially paired with a particular plugin package. If plugins are installed in a different package than the one the code which relies on the interface they provide, they will not be found when the application goes to load them.

4.2.4 Plugin Caching

In the course of using the Twisted plugin system, you may notice `dropin.cache` files appearing at various locations. These files are used to cache information about what plugins are present in the directory which contains them. At times, this cached information may become out of date. Twisted uses the `mtimes` of various files involved in the plugin system to determine when this cache may have become invalid. Twisted will try to re-write the cache each time it tries to use it but finds it out of date.

For a site-wide install, it may not (indeed, should not) be possible for applications running as normal users to rewrite the cache file. While these applications will still run and find correct plugin information, they may run more slowly than they would if the cache was up to date, and they may also report exceptions if certain plugins have been removed but which the cache still references. For these reasons, when installing or removing software which provides Twisted plugins, the site administrator should be sure the cache is regenerated. Well-behaved package managers for such software should take this task upon themselves, since it is trivially automatable. The canonical way to regenerate the cache is to run the following Python code:

```
from twisted.plugin import IPlugin, getPlugin
list(getPlugin(IPlugin))
```

As mentioned, it is normal for exceptions to be raised *once* here if plugins have been removed.

4.2.5 Further Reading

- *Components: Interfaces and Adapters* (page 144)

4.3 Writing a Twisted Application Plugin for twisted

This document describes writing extension subcommands for the `twisted` command, as a way to facilitate the deployment of your applications. (*This feature was added in Twisted 2.5*)

The target audience of this document are those that have developed a Twisted application which needs a command line-based deployment mechanism.

There are a few prerequisites to understanding this document:

- A basic understanding of the Twisted Plugin System (i.e., the `twisted.plugin` module) is necessary, however, step-by-step instructions will be given. Reading *The Twisted Plugin System* (page 140) is recommended, in particular the “Extending an Existing Program” section.
- The *Application* (page 155) infrastructure is used in Twisted Application Plugins; in particular, you should know how to expose your program’s functionality as a Service.
- In order to parse command line arguments, the Twisted Application Plugin system relies on `twisted.python.usage`, which is documented in *Using usage.Options* (page 159).

4.3.1 Goals

After reading this document, the reader should be able to expose their Service-using application as a subcommand of `twistd`, taking into consideration whatever was passed on the command line.

4.3.2 A note on .tap files

Readers may be confused about a historical file type associated with Twisted, the `.tap` file. This was a kind of file that was generated by a program named `mktag` and which `twistd` can read. `.tap` files are deprecated; this document has nothing to do with them, although the technology described herein is very closely related to the old system. “TAP”, in the modern Twisted vernacular, means “Twisted Application Plugin”.

4.3.3 Alternatives to TAP

The major alternative to the TAP mechanism is the `.tac` file, which is a simple script to be used with the `twistd -y/--python` parameter. The TAP plugin system exists to offer a more extensible command-line-driven interface to your application. For more information on `.tac` files, see the document *Using the Twisted Application Framework* (page 155).

4.3.4 Creating the plugin

The following directory structure is assumed of your project:

- *MyProject* - Top level directory
 - *myproject* - Python package
 - * `__init__.py`

During development of your project, Twisted plugins can be loaded from a special directory in your project, assuming your top level directory ends up in `sys.path`. Create a directory named `twisted` containing a directory named `plugins`, and add a file named `myproject.py` to it. This file will contain your plugin. Note that you should *not* add any `__init__.py` files to this directory structure.

In this file, define an object which *provides* the interfaces `twisted.plugin.IPlugin` and `twisted.application.service.IServiceMaker`.

The `tapname` attribute of your `IServiceMaker` provider will be used as the subcommand name in a command like `twistd [subcommand] [args...]`, and the `options` attribute (which should be a `usage.Options` subclass) will be used to parse the given args.

```
from zope.interface import implements

from twisted.python import usage
from twisted.plugin import IPlugin
from twisted.application.service import IServiceMaker
from twisted.application import internet

from myproject import MyFactory

class Options(usage.Options):
    optParameters = [("port", "p", 1235, "The port number to listen on.")]

class MyServiceMaker(object):
    implements(IServiceMaker, IPlugin)
    tapname = "myproject"
    description = "Run this! It'll make your dog happy."
    options = Options

    def makeService(self, options):
```



```

"""
Construct a TCPServer from a factory defined in myproject.
"""
return internet.TCPServer(int(options["port"]), MyFactory())

# Now construct an object which *provides* the relevant interfaces

# The name of this variable is irrelevant, as long as there is *some*
# name bound to a provider of IPlugin and IServiceMaker.

serviceMaker = MyServiceMaker()

```

Now running `twistd --help` should print `myproject` in the list of available subcommands, followed by the description that we specified in the plugin. `twistd -n myproject` would, assuming we defined a `MyFactory` factory inside `myproject`, start a listening server on port 1235 with that factory.

4.3.5 Conclusion

You should now be able to

- Create a `twistd` plugin
- Use it from your development environment
- Install it correctly and use it in deployment

4.4 Components: Interfaces and Adapters

Object oriented programming languages allow programmers to reuse portions of existing code by creating new “classes” of objects which subclass another class. When a class subclasses another, it is said to *inherit* all of its behaviour. The subclass can then “override” and “extend” the behavior provided to it by the superclass. Inheritance is very useful in many situations, but because it is so convenient to use, often becomes abused in large software systems, especially when multiple inheritance is involved. One solution is to use *delegation* instead of “inheritance” where appropriate. Delegation is simply the act of asking *another* object to perform a task for an object. To support this design pattern, which is often referred to as the *components* pattern because it involves many small interacting components, *interfaces* and *adapters* were created by the Zope 3 team.

“Interfaces” are simply markers which objects can use to say “I implement this interface”. Other objects may then make requests like “Please give me an object which implements interface X for object type Y”. Objects which implement an interface for another object type are called “adapters”.

The superclass-subclass relationship is said to be an *is-a* relationship. When designing object hierarchies, object modellers use subclassing when they can say that the subclass *is* the same class as the superclass. For example:

```

class Shape:
    sideLength = 0
    def getSideLength(self):
        return self.sideLength

    def setSideLength(self, sideLength):
        self.sideLength = sideLength

    def area(self):
        raise NotImplementedError, "Subclasses must implement area"

class Triangle(Shape):
    def area(self):
        return (self.sideLength * self.sideLength) / 2

class Square(Shape):

```

```
def area(self):
    return self.sideLength * self.sideLength
```

In the above example, a Triangle *is-a* Shape, so it subclasses Shape, and a Square *is-a* Shape, so it also subclasses Shape.

However, subclassing can get complicated, especially when Multiple Inheritance enters the picture. Multiple Inheritance allows a class to inherit from more than one base class. Software which relies heavily on inheritance often ends up having both very wide and very deep inheritance trees, meaning that one class inherits from many superclasses spread throughout the system. Since subclassing with Multiple Inheritance means *implementation inheritance*, locating a method's actual implementation and ensuring the correct method is actually being invoked becomes a challenge. For example:

```
class Area:
    sideLength = 0
    def getSideLength(self):
        return self.sideLength

    def setSideLength(self, sideLength):
        self.sideLength = sideLength

    def area(self):
        raise NotImplementedError, "Subclasses must implement area"

class Color:
    color = None
    def setColor(self, color):
        self.color = color

    def getColor(self):
        return self.color

class Square(Area, Color):
    def area(self):
        return self.sideLength * self.sideLength
```

The reason programmers like using implementation inheritance is because it makes code easier to read since the implementation details of Area are in a separate place than the implementation details of Color. This is nice, because conceivably an object could have a color but not an area, or an area but not a color. The problem, though, is that Square is not really an Area or a Color, but has an area and color. Thus, we should really be using another object oriented technique called *composition*, which relies on delegation rather than inheritance to break code into small reusable chunks. Let us continue with the Multiple Inheritance example, though, because it is often used in practice.

What if both the Color and the Area base class defined the same method, perhaps calculate? Where would the implementation come from? The implementation that is located for Square().calculate() depends on the method resolution order, or MRO, and can change when programmers change seemingly unrelated things by refactoring classes in other parts of the system, causing obscure bugs. Our first thought might be to change the calculate method name to avoid name clashes, to perhaps calculateArea and calculateColor. While explicit, this change could potentially require a large number of changes throughout a system, and is error-prone, especially when attempting to integrate two systems which you didn't write.

Let's imagine another example. We have an electric appliance, say a hair dryer. The hair dryer is american voltage. We have two electric sockets, one of them an american 110 Volt socket, and one of them a foreign 220 Volt socket. If we plug the hair dryer into the 220 Volt socket, it is going to expect 110 Volt current and errors will result. Going back and changing the hair dryer to support both plug110Volt and plug220Volt methods would be tedious, and what if we decided we needed to plug the hair dryer into yet another type of socket? For example:

```
class HairDryer:
    def plug(self, socket):
        if socket.voltage() == 110:
            print "I was plugged in properly and am operating."
        else:
```

```

        print "I was plugged in improperly and "
        print "now you have no hair dryer any more."

class AmericanSocket:
    def voltage(self):
        return 110

class ForeignSocket:
    def voltage(self):
        return 220

```

Given these classes, the following operations can be performed:

```

>>> hd = HairDryer()
>>> am = AmericanSocket()
>>> hd.plug(am)
I was plugged in properly and am operating.
>>> fs = ForeignSocket()
>>> hd.plug(fs)
I was plugged in improperly and
now you have no hair dryer any more.

```

We are going to attempt to solve this problem by writing an Adapter for the `ForeignSocket` which converts the voltage for use with an American hair dryer. An Adapter is a class which is constructed with one and only one argument, the “adaptee” or “original” object. In this example, we will show all code involved for clarity:

```

class AdaptToAmericanSocket:
    def __init__(self, original):
        self.original = original

    def voltage(self):
        return self.original.voltage() / 2

```

Now, we can use it as so:

```

>>> hd = HairDryer()
>>> fs = ForeignSocket()
>>> adapted = AdaptToAmericanSocket(fs)
>>> hd.plug(adapted)
I was plugged in properly and am operating.

```

So, as you can see, an adapter can ‘override’ the original implementation. It can also ‘extend’ the interface of the original object by providing methods the original object did not have. Note that an Adapter must explicitly delegate any method calls it does not wish to modify to the original, otherwise the Adapter cannot be used in places where the original is expected. Usually this is not a problem, as an Adapter is created to conform an object to a particular interface and then discarded.

4.4.1 Interfaces and Components in Twisted code

Adapters are a useful way of using multiple classes to factor code into discrete chunks. However, they are not very interesting without some more infrastructure. If each piece of code which wished to use an adapted object had to explicitly construct the adapter itself, the coupling between components would be too tight. We would like to achieve “loose coupling”, and this is where `twisted.python.components` comes in.

First, we need to discuss Interfaces in more detail. As we mentioned earlier, an Interface is nothing more than a class which is used as a marker. Interfaces should be subclasses of `zope.interface.Interface`, and have a very odd look to python programmers not used to them:

```

from zope.interface import Interface

class IAmericanSocket(Interface):

```

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

Notice how it looks just like a regular class definition, other than inheriting from `Interface`? However, the method definitions inside the class block do not have any method body! Since Python does not have any native language-level support for Interfaces like Java does, this is what distinguishes an Interface definition from a Class.

Now that we have a defined Interface, we can talk about objects using terms like this: “The `AmericanSocket` class implements the `IAmericanSocket` interface” and “Please give me an object which adapts `ForeignSocket` to the `IAmericanSocket` interface”. We can make *declarations* about what interfaces a certain class implements, and we can request adapters which implement a certain interface for a specific class.

Let’s look at how we declare that a class implements an interface:

```
from zope.interface import implements

class AmericanSocket:

    implements(IAmericanSocket)

    def voltage(self):
        return 110
```

So, to declare that a class implements an interface, we simply call `zope.interface.implements` at the class level. A single item tuple in Python is created by enclosing an item in parentheses and placing a single trailing comma after it.

Now, let’s say we want to rewrite the `AdaptToAmericanSocket` class as a real adapter. In this case we also specify it as implementing `IAmericanSocket`:

```
from zope.interface import implements

class AdaptToAmericanSocket:

    implements(IAmericanSocket)

    def __init__(self, original):
        """
        Pass the original ForeignSocket object as original
        """
        self.original = original

    def voltage(self):
        return self.original.voltage() / 2
```

Notice how we placed the `implements` declaration on this adapter class. So far, we have not achieved anything by using components other than requiring us to type more. In order for components to be useful, we must use the *component registry*. Since `AdaptToAmericanSocket` implements `IAmericanSocket` and regulates the voltage of a `ForeignSocket` object, we can *register* `AdaptToAmericanSocket` as an `IAmericanSocket` adapter for the `ForeignSocket` class. It is easier to see how this is done in code than to describe it:

```
from zope.interface import Interface, implements
from twisted.python import components

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

class AmericanSocket:
    implements(IAmericanSocket)
```

```

    def voltage(self):
        return 110

class ForeignSocket:
    def voltage(self):
        return 220

class AdaptToAmericanSocket:

    implements(IAmericanSocket)

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

    def voltage(self):
        return self.original.voltage() / 2

components.registerAdapter(
    AdaptToAmericanSocket,
    ForeignSocket,
    IAmericanSocket)

```

Now, if we run this script in the interactive interpreter, we can discover a little more about how to use components. The first thing we can do is discover whether an object implements an interface or not:

```

>>> IAmericanSocket.implementedBy(AmericanSocket)
True
>>> IAmericanSocket.implementedBy(ForeignSocket)
False
>>> as = AmericanSocket()
>>> fs = ForeignSocket()
>>> IAmericanSocket.providedBy(as)
True
>>> IAmericanSocket.providedBy(fs)
False

```

As you can see, the `AmericanSocket` instance claims to implement `IAmericanSocket`, but the `ForeignSocket` does not. If we wanted to use the `HairDryer` with the `AmericanSocket`, we could know that it would be safe to do so by checking whether it implements `IAmericanSocket`. However, if we decide we want to use `HairDryer` with a `ForeignSocket` instance, we must *adapt* it to `IAmericanSocket` before doing so. We use the interface object to do this:

```

>>> IAmericanSocket(fs)
<__main__.AdaptToAmericanSocket instance at 0x1a5120>

```

When calling an interface with an object as an argument, the interface looks in the adapter registry for an adapter which implements the interface for the given instance's class. If it finds one, it constructs an instance of the Adapter class, passing the constructor the original instance, and returns it. Now the `HairDryer` can safely be used with the adapted `ForeignSocket`. But what happens if we attempt to adapt an object which already implements `IAmericanSocket`? We simply get back the original instance:

```

>>> IAmericanSocket(as)
<__main__.AmericanSocket instance at 0x36bff0>

```

So, we could write a new “smart” `HairDryer` which automatically looked up an adapter for the socket you tried to plug it into:

```

class HairDryer:
    def plug(self, socket):

```

```

    adapted = IAmericanSocket(socket)
    assert adapted.voltage() == 110, "BOOM"
    print "I was plugged in properly and am operating"

```

Now, if we create an instance of our new “smart”`HairDryer` and attempt to plug it in to various sockets, the `HairDryer` will adapt itself automatically depending on the type of socket it is plugged in to:

```

>>> as = AmericanSocket()
>>> fs = ForeignSocket()
>>> hd = HairDryer()
>>> hd.plug(as)
I was plugged in properly and am operating
>>> hd.plug(fs)
I was plugged in properly and am operating

```

Voila; the magic of components.

Components and Inheritance

If you inherit from a class which implements some interface, and your new subclass declares that it implements another interface, the implements will be inherited by default.

For example, `pb.Root` (actually defined in `flavors.Root`) is a class which implements `IPBRoot`. This interface indicates that an object has remotely-invokable methods and can be used as the initial object served by a new `Broker` instance. It has an `implements` setting like:

```

from zope.interface import implements

class Root(Referenceable):
    implements(IPBRoot)

```

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

```

from zope.interface import implements, Interface

class IMyInterface(Interface):
    pass

class MyThing:
    implements(IMyInterface)

```

Now if you want to make this class inherit from `pb.Root`, the interfaces code will automatically determine that it also implements `IPBRoot`:

```

from twisted.spread import pb
from zope.interface import implements, Interface

class IMyInterface(Interface):
    pass

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

>>> from twisted.spread.flavors import IPBRoot
>>> IPBRoot.implementedBy(MyThing)
True

```

If you want `MyThing` to inherit from `pb.Root` but *not* implement `IPBRoot` like `pb.Root` does, use `implementOnly`:

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

class IMyInterface(Interface):
    pass

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

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

4.5 Cred: Pluggable Authentication

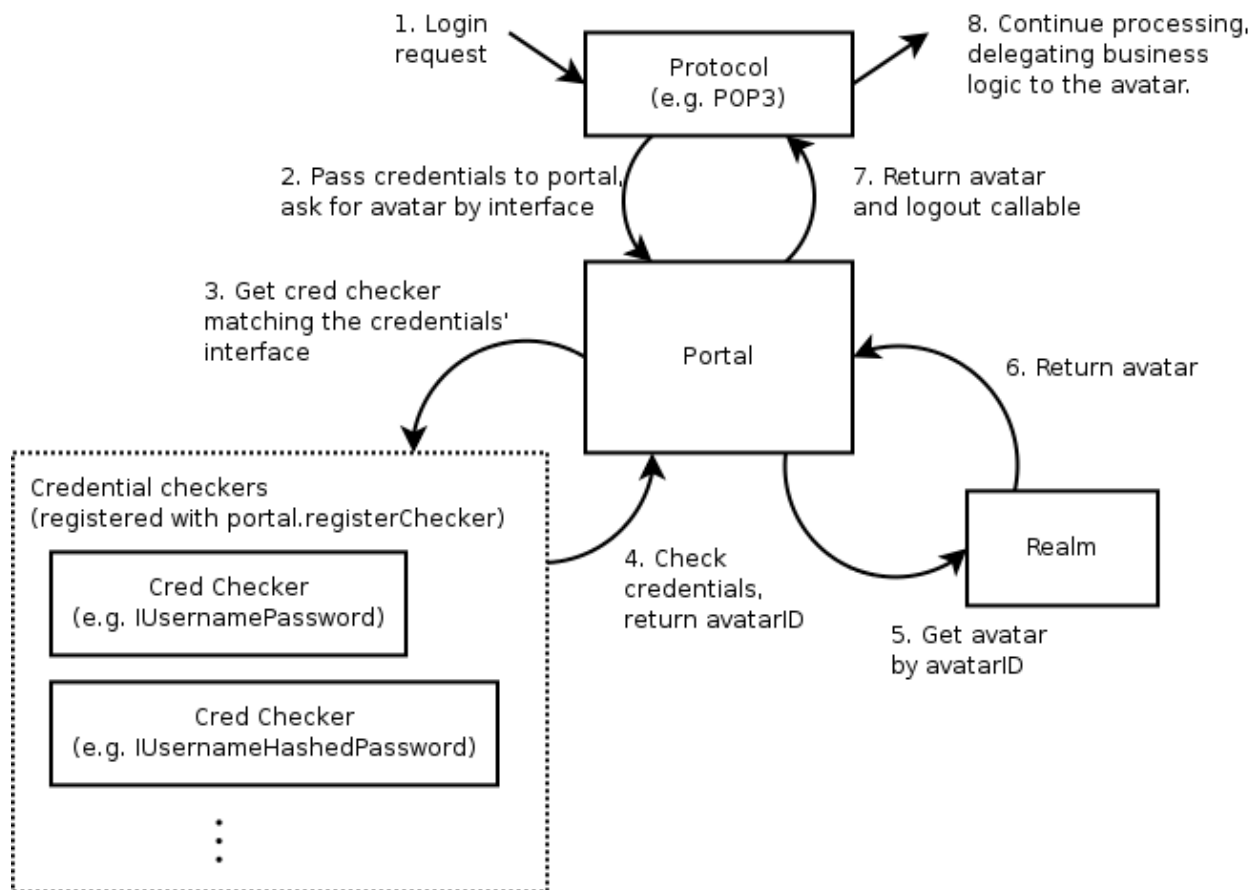
4.5.1 Goals

Cred is a pluggable authentication system for servers. It allows any number of network protocols to connect and authenticate to a system, and communicate to those aspects of the system which are meaningful to the specific protocol. For example, Twisted's POP3 support passes a "username and password" set of credentials to get back a mailbox for the specified email account. IMAP does the same, but retrieves a slightly different view of the same mailbox, enabling those features specific to IMAP which are not available in other mail protocols.

Cred is designed to allow both the backend implementation of the business logic - called the *avatar* - and the authentication database - called the *credential checker* - to be decided during deployment. For example, the same POP3 server should be able to authenticate against the local UNIX password database or an LDAP server without having to know anything about how or where mail is stored.

To sketch out how this works - a "Realm" corresponds to an application domain and is in charge of avatars, which are network-accessible business logic objects. To connect this to an authentication database, a top-level object called a *Portal* stores a realm, and a number of credential checkers. Something that wishes to log in, such as a *Protocol*, stores a reference to the portal. Login consists of passing credentials and a request interface (e.g. POP3's *IMailbox*) to the portal. The portal passes the credentials to the appropriate credential checker, which returns an avatar ID. The ID is passed to the realm, which returns the appropriate avatar. For a Portal that has a realm that creates mailbox objects and a credential checker that checks `/etc/passwd`, login consists of passing in a username/password and the *IMailbox* interface to the portal. The portal passes this to the `/etc/passwd` credential checker, gets back a avatar ID corresponding to an email account, passes that to the realm and gets back a mailbox object for that email account.

Putting all this together, here's how a login request will typically be processed:



4.5.2 Cred objects

The Portal

This is the the core of login, the point of integration between all the objects in the cred system. There is one concrete implementation of `Portal`, and no interface - it does a very simple task. A `Portal` associates one (1) `Realm` with a collection of `CredentialChecker` instances. (More on those later.)

If you are writing a protocol that needs to authenticate against something, you will need a reference to a `Portal`, and to nothing else. This has only 2 methods -

- `login(credentials, mind, *interfaces)`

The docstring is quite expansive (see `twisted.cred.portal`), but in brief, this is what you call when you need to call in order to connect a user to the system. Typically you only pass in one interface, and the `mind` is `None`. The interfaces are the possible interfaces the returned avatar is expected to implement, in order of preference. The result is a deferred which fires a tuple of:

- interface the avatar implements (which was one of the interfaces passed in the `*interfaces` tuple)
- an object that implements that interface (an avatar)
- `logout`, a 0-argument callable which disconnects the connection that was established by this call to `login`

The `logout` method has to be called when the avatar is logged out. For POP3 this means when the protocol is disconnected or logged out, etc..

- `registerChecker(checker, *credentialInterfaces)`

which adds a `CredentialChecker` to the portal. The optional list of interfaces are interfaces of credentials that the checker is able to check.

The CredentialChecker

This is an object implementing `ICredentialsChecker` which resolves some Credentials to an avatar ID. Some examples of CredentialChecker implementations would be: `InMemoryUsernamePassword`, `ApacheStyleHTAccessFile`, `UNIXPasswordDatabase`, `SSHPublicKeyDatabase`. A credential checker stipulates some requirements of the credentials it can check by specifying a `credentialInterfaces` attribute, which is a list of interfaces. Credentials passed to its `requestAvatarId` method must implement one of those interfaces.

For the most part, these things will just check usernames and passwords and produce the username as the result, but hopefully we will be seeing some public-key, challenge-response, and certificate based credential checker mechanisms soon.

A credential checker should raise an error if it cannot authenticate the user, and return `twisted.cred.checkers.ANONYMOUS` for anonymous access.

The Credentials

Oddly enough, this represents some credentials that the user presents. Usually this will just be a small static blob of data, but in some cases it will actually be an object connected to a network protocol. For example, a username/password pair is static, but a challenge/response server is an active state-machine that will require several method calls in order to determine a result.

Twisted comes with a number of credentials interfaces and implementations in the `twisted.cred.credentials` module, such as `IUsernamePassword` and `IUsernameHashedPassword`.

The Realm

A realm is an interface which connects your universe of “business objects” to the authentication system.

`IRrealm` is another one-method interface:

- `requestAvatar(avatarId, mind, *interfaces)`

This method will typically be called from `'Portal.login'`. The `avatarId` is the one returned by a `CredentialChecker`.

Note: Note that `avatarId` must always be a string. In particular, do not use unicode strings. If internationalized support is needed, it is recommended to use UTF-8, and take care of decoding in the realm.

The important thing to realize about this method is that if it is being called, *the user has already authenticated*. Therefore, if possible, the Realm should create a new user if one does not already exist whenever possible. Of course, sometimes this will be impossible without more information, and that is the case that the `interfaces` argument is for.

Since `requestAvatar` should be called from a Deferred callback, it may return a Deferred or a synchronous result.

The Avatar

An avatar is a business logic object for a specific user. For POP3, it's a mailbox, for a first-person-shooter it's the object that interacts with the game, the actor as it were. Avatars are specific to an application, and each avatar represents a single “user”.

The Mind

As mentioned before, the mind is usually `None`, so you can skip this bit if you want.

Masters of Perspective Broker already know this object as the ill-named “client object”. There is no “mind” class, or even interface, but it is an object which serves an important role - any notifications which are to be relayed to an authenticated client are passed through a ‘mind’. In addition, it allows passing more information to the realm during login in addition to the avatar ID.

The name may seem rather unusual, but considering that a Mind is representative of the entity on the “other end” of a network connection that is both receiving updates and issuing commands, I believe it is appropriate.

Although many protocols will not use this, it serves an important role. It is provided as an argument both to the `Portal` and to the `Realm`, although a `CredentialChecker` should interact with a client program exclusively through a `Credentials` instance.

Unlike the original Perspective Broker “client object”, a Mind’s implementation is most often dictated by the protocol that is connecting rather than the Realm. A Realm which requires a particular interface to issue notifications will need to wrap the Protocol’s mind implementation with an adapter in order to get one that conforms to its expected interface - however, Perspective Broker will likely continue to use the model where the client object has a pre-specified remote interface.

(If you don’t quite understand this, it’s fine. It’s hard to explain, and it’s not used in simple usages of cred, so feel free to pass None until you find yourself requiring something like this.)

4.5.3 Responsibilities

Server protocol implementation

The protocol implementor should define the interface the avatar should implement, and design the protocol to have a portal attached. When a user logs in using the protocol, a credential object is created, passed to the portal, and an avatar with the appropriate interface is requested. When the user logs out or the protocol is disconnected, the avatar should be logged out.

The protocol designer should not hardcode how users are authenticated or the realm implemented. For example, a POP3 protocol implementation would require a portal whose realm returns avatars implementing IMailbox and whose credential checker accepts username/password credentials, but that is all. Here’s a sketch of how the code might look - note that USER and PASS are the protocol commands used to login, and the DELE command can only be used after you are logged in:

```
from zope.interface import Interface

from twisted.protocols import basic
from twisted.python import log
from twisted.cred import credentials, error
from twisted.internet import defer

class IMailbox(Interface):
    """Interface specification for mailbox."""
    def deleteMessage(index): pass

class POP3(basic.LineReceiver):
    # ...
    def __init__(self, portal):
        self.portal = portal

    def do_DELE(self, i):
        # uses self.mbox, which is set after login
        i = int(i)-1
        self.mbox.deleteMessage(i)
        self.successResponse()

    def do_USER(self, user):
        self._userIs = user
        self.successResponse('USER accepted, send PASS')

    def do_PASS(self, password):
        if self._userIs is None:
            self.failResponse("USER required before PASS")
            return
        user = self._userIs
        self._userIs = None
        d = defer.maybeDeferred(self.authenticateUserPASS, user, password)
        d.addCallback(self._cbMailbox, user)
```

```

def authenticateUserPASS(self, user, password):
    if self.portal is not None:
        return self.portal.login(
            cred.credentials.UsernamePassword(user, password),
            None,
            IMailbox
        )
    raise error.UnauthorizedLogin()

def _cbMailbox(self, ial, user):
    interface, avatar, logout = ial

    if interface is not IMailbox:
        self.failResponse('Authentication failed')
        log.err("_cbMailbox() called with an interface other than IMailbox")
        return

    self.mbox = avatar
    self._onLogout = logout
    self.successResponse('Authentication succeeded')
    log.msg("Authenticated login for " + user)

```

Application implementation

The application developer can implement realms and credential checkers. For example, she might implement a realm that returns IMailbox implementing avatars, using MySQL for storage, or perhaps a credential checker that uses LDAP for authentication. In the following example, the Realm for a simple remote object service (using Twisted's Perspective Broker protocol) is implemented:

```

from twisted.spread import pb
from twisted.cred.portal import IRealm

class SimplePerspective(pb.Avatar):

    def perspective_echo(self, text):
        print 'echoing',text
        return text

    def logout(self):
        print self, "logged out"

class SimpleRealm:
    implements(IRrealm)

    def requestAvatar(self, avatarId, mind, *interfaces):
        if pb.IPerspective in interfaces:
            avatar = SimplePerspective()
            return pb.IPerspective, avatar, avatar.logout
        else:
            raise NotImplementedError("no interface")

```

Deployment

Deployment involves tying together a protocol, an appropriate realm and a credential checker. For example, a POP3 server can be constructed by attaching to it a portal that wraps the MySQL-based realm and an /etc/passwd credential checker, or perhaps the LDAP credential checker if that is more useful. The following example shows how the SimpleRealm in the previous example is deployed using an in-memory credential checker:

```

from twisted.spread import pb
from twisted.internet import reactor
from twisted.cred.portal import Portal
from twisted.cred.checkers import InMemoryUsernamePasswordDatabaseDontUse

portal = Portal(SimpleRealm())
checker = InMemoryUsernamePasswordDatabaseDontUse()
checker.addUser("guest", "password")
portal.registerChecker(checker)
reactor.listenTCP(9986, pb.PBServerFactory(portal))
reactor.run()

```

4.6 Using the Twisted Application Framework

4.6.1 Introduction

Audience

The target audience of this document is a Twisted user who wants to deploy a significant amount of Twisted code in a re-usable, standard and easily configurable fashion. A Twisted user who wishes to use the Application framework needs to be familiar with developing Twisted *servers* (page 13) and/or *clients* (page 17).

Goals

- To introduce the Twisted Application infrastructure.
- To explain how to deploy your Twisted application using `.tac` files and `twistd`
- To outline the existing Twisted services.

4.6.2 Overview

The Twisted Application infrastructure takes care of running and stopping your application. Using this infrastructure frees you from having to write a large amount of boilerplate code by hooking your application into existing tools that manage daemonization, logging, *choosing a reactor* (page 135) and more.

The major tool that manages Twisted applications is a command-line utility called `twistd`. `twistd` is cross platform, and is the recommended tool for running Twisted applications.

The core component of the Twisted Application infrastructure is the `twisted.application.service.Application` object — an object which represents your application. However, `Application` doesn't provide anything that you'd want to manipulate directly. Instead, `Application` acts as a container of any “Services” (objects implementing `IService`) that your application provides. Most of your interaction with the Application infrastructure will be done through `Services`.

By “Service”, we mean anything in your application that can be started and stopped. Typical services include web servers, FTP servers and SSH clients. Your `Application` object can contain many services, and can even contain structured heirarchies of `Services` using `IServiceCollections`.

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

```

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

port = 7001
factory = EchoFactory()

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

```

See *Writing Servers* (page 13) for an explanation of EchoFactory.
This example creates a simple heirarchy:

```
application
|
`- echoService
```

More complicated heirarchies of services can be created using IServiceCollection. You will most likely want to do this to manage Services which are dependent on other Services. For example, a proxying Twisted application might want its server Service to only start up after the associated Client service.

4.6.3 Using application

twistd and tac

To handle start-up and configuration of your Twisted application, the Twisted Application infrastructure uses .tac files. .tac are Python files which configure an Application object and assign this object to the top-level variable “application”.

The following is a simple example of a .tac file:

```
"""
This is an example .tac file which starts a webserver on port 8080 and
serves files from the current working directory.

The important part of this, the part that makes it a .tac file, is
the final root-level section, which sets up the object called 'application'
which twistd will look for
"""

import os
from twisted.application import service, internet
from twisted.web import static, server

def getWebService():
    """
    Return a service suitable for creating an application object.

    This service is a simple web server that serves files on port 8080 from
    underneath the current working directory.
    """
    # create a resource to serve static files
    fileServer = server.Site(static.File(os.getcwd()))
    return internet.TCPServer(8080, fileServer)

# this is the core part of any tac file, the creation of the root-level
# application object
application = service.Application("Demo application")

# attach the service to its parent application
service = getWebService()
service.setServiceParent(application)
```

Source listing — *service.tac*

twistd is a program that runs Twisted applications using a .tac file. In its most simple form, it takes a single argument -y and a tac file name. For example, you can run the above server with the command `twistd -y service.tac`.

By default, `twistd` daemonizes and logs to a file called `twistd.log`. More usually, when debugging, you will want your application to run in the foreground and log to the command line. To run the above file like this, use the command `twistd -noy service.tac`

For more information, see the `twistd` man page.

Services provided by Twisted

Twisted provides several services that you want to know about.

Each of these services (except `TimerService`) has a corresponding “connect” or “listen” method on the reactor, and the constructors for the services take the same arguments as the reactor methods. The “connect” methods are for clients and the “listen” methods are for servers. For example, `TCPServer` corresponds to `reactor.listenTCP` and `TCPClient` corresponds to `reactor.connectTCP`.

TCPServer

TCPClient Services which allow you to make connections and listen for connections on TCP ports.

- `listenTCP`
- `connectTCP`

UNIXServer

UNIXClient Services which listen and make connections over UNIX sockets.

- `listenUNIX`
- `connectUNIX`

SSLServer

SSLClient Services which allow you to make SSL connections and run SSL servers.

- `listenSSL`
- `connectSSL`

UDPServer

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

- `listenUDP`
- `connectUDP`

See also the *UDP documentation* (page 90).

UNIXDatagramServer

UNIXDatagramClient Services which send and receive data over UNIX datagram sockets.

- `listenUNIXDatagram`
- `connectUNIXDatagram`

MulticastServer A server for UDP socket methods that support multicast.

- `listenMulticast`

TimerService A service to periodically call a function.

Service Collection

`IServiceCollection` objects contain `IService` objects. `IService` objects can be added to `IServiceCollection` by calling `setServiceParent` and detached by using `disownServiceParent`.

The standard implementation of `IServiceCollection` is `MultiService`, which also implements `IService`. `MultiService` is useful for creating a new `Service` which combines two or more existing `Services`. For example, you could create a `DNS Service` as a `MultiService` which has a `TCP` and a `UDP Service` as children.

```
from twisted.application import internet, service
from twisted.names import server, dns, hosts

port = 53

# Create a MultiService, and hook up a TCPServer and a UDPServer to it as
# children.
dnsService = service.MultiService()
hostsResolver = hosts.Resolver('/etc/hosts')
tcpFactory = server.DNSServerFactory([hostsResolver])
internet.TCPServer(port, tcpFactory).setServiceParent(dnsService)
udpFactory = dns.DNSDatagramProtocol(tcpFactory)
internet.UDPServer(port, udpFactory).setServiceParent(dnsService)

# Create an application as normal
application = service.Application("DNSExample")

# Connect our MultiService to the application, just like a normal service.
dnsService.setServiceParent(application)
```

Chapter 5

Utilities

5.1 Using `usage.Options`

5.1.1 Introduction

There is frequently a need for programs to parse a UNIX-like command line program: options preceded by `-` or `--`, sometimes followed by a parameter, followed by a list of arguments. The `twisted.python.usage` provides a class, `Options`, to facilitate such parsing.

While Python has the `getopt` module for doing this, it provides a very low level of abstraction for options. Twisted has a higher level of abstraction, in the class `twisted.python.usage.Options`. It uses Python's reflection facilities to provide an easy to use yet flexible interface to the command line. While most command line processors either force the application writer to write her own loops, or have arbitrary limitations on the command line (the most common one being not being able to have more than one instance of a specific option, thus rendering the idiom `program -v -v -v` impossible), Twisted allows the programmer to decide how much control she wants.

The `Options` class is used by subclassing. Since a lot of time it will be used in the `twisted.tap` package, where the local conventions require the specific options parsing class to also be called `Options`, it is usually imported with

```
from twisted.python import usage
```

5.1.2 Boolean Options

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

```
class Options(usage.Options):

    optFlags = [{"fast", "f", "Act quickly"}, {"safe", "s", "Act safely"}]
```

`optFlags` should be a list of 3-lists. The first element is the long name, and will be used on the command line as `--fast`. The second one is the short name, and will be used on the command line as `-f`. The last element is a description of the flag and will be used to generate the usage information text. The long name also determines the name of the key that will be set on the `Options` instance. Its value will be 1 if the option was seen, 0 otherwise. Here is an example for usage:

```
class Options(usage.Options):

    optFlags = [
        ["fast", "f", "Act quickly"],
        ["good", "g", "Act well"],
        ["cheap", "c", "Act cheaply"]
    ]

    command_line = ["-g", "--fast"]

    options = Options()
```



```

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

```

The above will print `fast good`.

Note here that Options fully supports the mapping interface. You can access it mostly just like you can access any other dict. Options are stored as mapping items in the Options instance: parameters as 'paramname': 'value' and flags as 'flagname': 1 or 0.

Inheritance, Or: How I Learned to Stop Worrying and Love the Superclass

Sometimes there is a need for several option processors with a unifying core. Perhaps you want all your commands to understand `-q/--quiet` means to be quiet, or something similar. On the face of it, this looks impossible: in Python, the subclass's `optFlags` would shadow the superclass's. However, `usage.Options` uses special reflection code to get all of the `optFlags` defined in the hierarchy. So the following:

```

class BaseOptions(usage.Options):

    optFlags = [{"quiet", "q", None}]

class SpecificOptions(BaseOptions):

    optFlags = [
        ["fast", "f", None], ["good", "g", None], ["cheap", "c", None]
    ]

```

Is the same as:

```

class SpecificOptions(BaseOptions):

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

```

5.1.3 Parameters

Parameters are specified using the attribute `optParameters`. They *must* be given a default. If you want to make sure you got the parameter from the command line, give a non-string default. Since the command line only has strings, this is completely reliable.

Here is an example:

```

from twisted.python import usage

class Options(usage.Options):

    optFlags = [

```

```

        ["fast", "f", "Run quickly"],
        ["good", "g", "Don't validate input"],
        ["cheap", "c", "Use cheap resources"]
    ]
    optParameters = [["user", "u", None, "The user name"]]

config = Options()
try:
    config.parseOptions() # When given no argument, parses sys.argv[1:]
except usage.UsageError, errortext:
    print '%s: %s' % (sys.argv[0], errortext)
    print '%s: Try --help for usage details.' % (sys.argv[0])
    sys.exit(1)

if config['user'] is not None:
    print "Hello", config['user']
print "So, you want it:"

if config['fast']:
    print "fast",
if config['good']:
    print "good",
if config['cheap']:
    print "cheap",
print

```

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

5.1.4 Option Subcommands

It is useful, on occasion, to group a set of options together based on the logical “action” to which they belong. For this, the `usage.Options` class allows you to define a set of “subcommands”, each of which can provide its own `usage.Options` instance to handle its particular options.

Here is an example for an `Options` class that might parse options like those the `cvs` program takes

```

from twisted.python import usage

class ImportOptions(usage.Options):
    optParameters = [
        ['module', 'm', None, None], ['vendor', 'v', None, None],
        ['release', 'r', None]
    ]

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

class Options(usage.Options):
    subCommands = [['import', None, ImportOptions, "Do an Import"],
                   ['checkout', None, CheckoutOptions, "Do a Checkout"]]

    optParameters = [
        ['compression', 'z', 0, 'Use compression'],
        ['repository', 'r', None, 'Specify an alternate repository']
    ]

config = Options(); config.parseOptions()
if config.subCommand == 'import':
    doImport(config.subOptions)

```

```
elif config.subCommand == 'checkout':
    doCheckout(config.subOptions)
```

The `subCommands` attribute of `Options` directs the parser to the two other `Options` subclasses when the strings "import" or "checkout" are present on the command line. All options after the given command string are passed to the specified `Options` subclass for further parsing. Only one subcommand may be specified at a time. After parsing has completed, the `Options` instance has two new attributes - `subCommand` and `subOptions` - which hold the command string and the `Options` instance used to parse the remaining options.

5.1.5 Generic Code For Options

Sometimes, just setting an attribute on the basis of the options is not flexible enough. In those cases, Twisted does not even attempt to provide abstractions such as "counts" or "lists", but rather lets you call your own method, which will be called whenever the option is encountered.

Here is an example of counting verbosity

```
from twisted.python import usage

class Options(usage.Options):

    def __init__(self):
        usage.Options.__init__(self)
        self['verbosity'] = 0 # default

    def opt_verbose(self):
        self['verbosity'] = self['verbosity']+1

    def opt_quiet(self):
        self['verbosity'] = self['verbosity']-1

    opt_v = opt_verbose
    opt_q = opt_quiet
```

Command lines that look like command `-v -v -v -v` will increase verbosity to 4, while command `-q -q -q` will decrease verbosity to -3.

The `usage.Options` class knows that these are parameter-less options, since the methods do not receive an argument. Here is an example for a method with a parameter:

```
from twisted.python import usage

class Options(usage.Options):

    def __init__(self):
        usage.Options.__init__(self)
        self['symbols'] = []

    def opt_define(self, symbol):
        self['symbols'].append(symbol)

    opt_D = opt_define
```

This example is useful for the common idiom of having command `-DFOO -DBAR` to define symbols.

5.1.6 Parsing Arguments

`usage.Options` does not stop helping when the last parameter is gone. All the other arguments are sent into a function which should deal with them. Here is an example for a `cmp` like command.

```

from twisted.python import usage

class Options(usage.Options):

    optParameters = [{"max_differences", "d", 1, None}]

    def parseArgs(self, origin, changed):
        self['origin'] = origin
        self['changed'] = changed

```

The command should look like `command origin changed`.

If you want to have a variable number of left-over arguments, just use `def parseArgs(self, *args):`. This is useful for commands like the UNIX `cat(1)`.

5.1.7 Post Processing

Sometimes, you want to perform post processing of options to patch up inconsistencies, and the like. Here is an example:

```

from twisted.python import usage

class Options(usage.Options):

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

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

```

5.2 Logging with twisted.python.log

5.2.1 Basic usage

Twisted provides a simple and flexible logging system in the `twisted.python.log` module. It has three commonly used functions:

msg Logs a new message. For example:

```

from twisted.python import log
log.msg('Hello, world.')

```

err Writes a failure to the log, including traceback information (if any). You can pass it a `Failure` or `Exception` instance, or nothing. If you pass something else, it will be converted to a string with `repr` and logged. If you pass nothing, it will construct a `Failure` from the currently active exception, which makes it convenient to use in an `except` clause:

```

try:
    x = 1 / 0
except:
    log.err()    # will log the ZeroDivisionError

```

startLogging Starts logging to a given file-like object. For example:

```

log.startLogging(open('/var/log/foo.log', 'w'))

```

or:

```
log.startLogging(sys.stdout)
```

By default, `startLogging` will also redirect anything written to `sys.stdout` and `sys.stderr` to the log. You can disable this by passing `setStdout=False` to `startLogging`.

Before `startLogging` is called, log messages will be discarded and errors will be written to `stderr`.

Logging and twistd

If you are using `twistd` to run your daemon, it will take care of calling `startLogging` for you, and will also rotate log files. See *twistd and tac* (page 156) and the `twistd` man page for details of using `twistd`.

Log files

The `twisted.python.logfile` module provides some standard classes suitable for use with `startLogging`, such as `DailyLogFile`, which will rotate the log to a new file once per day.

5.2.2 Writing log observers

Log observers are the basis of the Twisted logging system. An example of a log observer in Twisted is the `FileLogObserver` used by `startLogging` that writes events to a log file. A log observer is just a callable that accepts a dictionary as its only argument. You can then register it to receive all log events (in addition to any other observers):

```
twisted.python.log.addObserver(yourCallable)
```

The dictionary will have at least two items:

message The message (a list, usually of strings) for this log event, as passed to `log.msg` or the message in the failure passed to `log.err`.

isError This is a boolean that will be true if this event came from a call to `log.err`. If this is set, there may be a `failure` item in the dictionary as well, with a `Failure` object in it.

Other items the built in logging functionality may add include:

printed This message was captured from `sys.stdout`, i.e. this message came from a `print` statement. If `isError` is also true, it came from `sys.stderr`.

You can pass additional items to the event dictionary by passing keyword arguments to `log.msg` and `log.err`. The standard log observers will ignore dictionary items they don't use.

Important notes:

- Never raise an exception from a log observer. If your log observer raises an exception, it will be removed.
- Never block in a log observer, as it may run in main Twisted thread. This means you can't use socket or syslog Python-logging backends.
- The observer needs to be thread safe if you anticipate using threads in your program.

5.3 DirDBM: Directory-based Storage

5.3.1 dirdbm.DirDBM

`twisted.persisted.dirdbm.DirDBM` is a DBM-like storage system. That is, it stores mappings between keys and values, like a Python dictionary, except that it stores the values in files in a directory - each entry is a different file. The keys must always be strings, as are the values. Other than that, `DirDBM` objects act just like Python dictionaries.

`DirDBM` is useful for cases when you want to store small amounts of data in an organized fashion, without having to deal with the complexity of a RDBMS or other sophisticated database. It is simple, easy to use, cross-platform, and doesn't require any external C libraries, unlike Python's built-in DBM modules.

```
>>> from twisted.persisted import dirdbm
>>> d = dirdbm.DirDBM("/tmp/dir")
>>> d["librarian"] = "ook"
>>> d["librarian"]
'ook'
>>> d.keys()
['librarian']
>>> del d["librarian"]
>>> d.items()
[]
```

5.3.2 dirdbm.Shelf

Sometimes it is necessary to persist more complicated objects than strings. With some care, `dirdbm.Shelf` can transparently persist them. `Shelf` works exactly like `DirDBM`, except that the values (but not the keys) can be arbitrary picklable objects. However, notice that mutating an object after it has been stored in the `Shelf` has no effect on the `Shelf`. When mutating objects, it is necessary to explicitly store them back in the `Shelf` afterwards:

```
>>> from twisted.persisted import dirdbm
>>> d = dirdbm.Shelf("/tmp/dir2")
>>> d["key"] = [1, 2]
>>> d["key"]
[1, 2]
>>> l = d["key"]
>>> l.append(3)
>>> d["key"]
[1, 2]
>>> d["key"] = l
>>> d["key"]
[1, 2, 3]
```

5.4 Using telnet to manipulate a twisted server

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

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

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

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

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

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

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

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

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

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

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

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

5.5 Writing tests for Twisted code

5.5.1 Trial basics

Trial is Twisted's testing framework. It provides a library for writing test cases and utility functions for working with the Twisted environment in your tests, and a command-line utility for running your tests. Trial is built on the Python standard library's `unittest` module.

To run all the Twisted tests, do:

```
$ trial twisted
```

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

5.5.2 Twisted-specific quirks: reactor, Deferreds, callLater

The standard Python `unittest` framework, from which Trial is derived, is ideal for testing code with a fairly linear flow of control. Twisted is an asynchronous networking framework which provides a clean, sensible way to establish functions that are run in response to events (like timers and incoming data), which creates a highly non-linear flow of control. Trial has a few extensions which help to test this kind of code. This section provides some hints on how to use these extensions and how to best structure your tests.

Leave the Reactor as you found it

Trial runs the entire test suite (over two thousand tests) in a single process, with a single reactor. Therefore it is important that your test leave the reactor in the same state as it found it. Leftover timers may expire during somebody else's unsuspecting test. Leftover connection attempts may complete (and fail) during a later test. These lead to intermittent failures that wander from test to test and are very time-consuming to track down.

Your test is responsible for cleaning up after itself. The `tearDown` method is an ideal place for this cleanup code: it is always run regardless of whether your test passes or fails (like a bare `except` clause in a try-except construct). Exceptions in `tearDown` are flagged as errors and flunk the test.

If your code uses Deferreds or depends on the reactor running, you can return a Deferred from your test method, `setUp`, or `tearDown` and Trial will do the right thing. That is, it will run the reactor for you until the Deferred has triggered and its callbacks have been run. Don't use `reactor.run()`, `reactor.stop()`, or `reactor.iterate()` in your tests.

Calls to `reactor.callLater` create `IDelayedCalls`. These need to be run or cancelled during a test, otherwise they will outlive the test. This would be bad, because they could interfere with a later test, causing confusing failures in unrelated tests! For this reason, Trial checks the reactor to make sure there are no leftover `IDelayedCalls` in the reactor after a test, and will fail the test if there are. The cleanest and simplest way to make sure this all works is to return a `Deferred` from your test.

Similarly, sockets created during a test should be closed by the end of the test. This applies to both listening ports and client connections. So, calls to `reactor.listenTCP` (and `listenUNIX`, and so on) return `IListeningPorts`, and these should be cleaned up before a test ends by calling their `stopListening` method. Calls to `reactor.connectTCP` return `IConnectors`, which should be cleaned up by calling their `disconnect` method. Trial will warn about unclosed sockets.

The golden rule is: If your tests call a function which returns a `Deferred`, your test should return a `Deferred`.

Using Timers to Detect Failing Tests

It is common for tests to establish some kind of fail-safe timeout that will terminate the test in case something unexpected has happened and none of the normal test-failure paths are followed. This timeout puts an upper bound on the time that a test can consume, and prevents the entire test suite from stalling because of a single test. This is especially important for the Twisted test suite, because it is run automatically by the buildbot whenever changes are committed to the Subversion repository.

The way to do this in Trial is to set the `.timeout` attribute on your unit test method. Set the attribute to the number of seconds you wish to elapse before the test raises a timeout error.

Chapter 6

Twisted RDBMS support

6.1 twisted.enterprise.adbapi: Twisted RDBMS support

6.1.1 Abstract

Twisted is an asynchronous networking framework, but most database API implementations unfortunately have blocking interfaces – for this reason, `twisted.enterprise.adbapi` was created. It is a non-blocking interface to the standardized DB-API 2.0 API, which allows you to access a number of different RDBMSes.

6.1.2 What you should already know

- Python :-)
- How to write a simple Twisted Server (see *this tutorial* (page 13) to learn how)
- Familiarity with using database interfaces (see the documentation for DBAPI 2.0¹ or this article² by Andrew Kuchling)

6.1.3 Quick Overview

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

```
# Create connection...
db = dbmodule.connect('mydb', 'andrew', 'password')
# ...which blocks for an unknown amount of time

# Create a cursor
cursor = db.cursor()

# Do a query...
resultset = cursor.query('SELECT * FROM table WHERE ...')
# ...which could take a long time, perhaps even minutes.
```

Those delays are unacceptable when using an asynchronous framework such as Twisted. For this reason, twisted provides `twisted.enterprise.adbapi`, an asynchronous wrapper for any DB-API 2.0³-compliant module.

`enterprise.adbapi` will do blocking database operations in separate threads, which trigger callbacks in the originating thread when they complete. In the meantime, the original thread can continue doing normal work, like servicing other requests.

¹<http://www.python.org/topics/database/DatabaseAPI-2.0.html>

²<http://www.amk.ca/python/writing/DB-API.html>

³<http://www.python.org/topics/database/DatabaseAPI-2.0.html>

6.1.4 How do I use adbapi?

Rather than creating a database connection directly, use the `adbapi.ConnectionPool` class to manage a connections for you. This allows `enterprise.adbapi` to use multiple connections, one per thread. This is easy:

```
# Using the "dbmodule" from the previous example, create a ConnectionPool
from twisted.enterprise import adbapi
dbpool = adbapi.ConnectionPool("dbmodule", 'mydb', 'andrew', 'password')
```

Things to note about doing this:

- There is no need to import `dbmodule` directly. You just pass the name to `adbapi.ConnectionPool`'s constructor.
- The parameters you would pass to `dbmodule.connect` are passed as extra arguments to `adbapi.ConnectionPool`'s constructor. Keyword parameters work as well.

Now we can do a database query:

```
# equivalent of cursor.execute(statement), return cursor.fetchall():
def getAge(user):
    return dbpool.runQuery("SELECT age FROM users WHERE name = ?", user)

def printResult(l):
    if l:
        print l[0][0], "years old"
    else:
        print "No such user"

getAge("joe").addCallback(printResult)
```

This is straightforward, except perhaps for the return value of `getAge`. It returns a `twisted.internet.defer.Deferred`, which allows arbitrary callbacks to be called upon completion (or upon failure). More documentation on `Deferred` is available *here* (page 99).

In addition to `runQuery`, there is also `runOperation`, and `runInteraction` that gets called with a callable (e.g. a function). The function will be called in the thread with a `twisted.enterprise.adbapi.Transaction`, which basically mimics a DB-API cursor. In all cases a database transaction will be committed after your database usage is finished, unless an exception is raised in which case it will be rolled back.

```
def _getAge(txn, user):
    # this will run in a thread, we can use blocking calls
    txn.execute("SELECT * FROM foo")
    # ... other cursor commands called on txn ...
    txn.execute("SELECT age FROM users WHERE name = ?", user)
    result = txn.fetchall()
    if result:
        return result[0][0]
    else:
        return None

def getAge(user):
    return dbpool.runInteraction(_getAge, user)

def printResult(age):
    if age != None:
        print age, "years old"
    else:
        print "No such user"

getAge("joe").addCallback(printResult)
```

Also worth noting is that these examples assumes that dbmodule uses the “qmarks” paramstyle (see the DB-API specification). If your dbmodule uses a different paramstyle (e.g. pyformat) then use that. Twisted doesn’t attempt to offer any sort of magic paramater munging – `runQuery(query, params, ...)` maps directly onto `cursor.execute(query, params, ...)`.

6.1.5 Examples of various database adapters

Notice that the first argument is the module name you would usually import and get `connect(...)` from, and that following arguments are whatever arguments you’d call `connect(...)` with.

```
from twisted.enterprise import adbapi

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

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

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

6.1.6 And that’s it!

That’s all you need to know to use a database from within Twisted. You probably should read the adbapi module’s documentation to get an idea of the other functions it has, but hopefully this document presents the core ideas.

6.2 Twisted Enterprise Row Objects

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

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

If row does not fit your model, you will be best off using the low-level database API (page 168) directly, or writing your own object/relational layer on top of it.

6.2.1 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, [(child ColumnName, 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.

6.2.2 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] )
```

6.2.3 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.

6.2.4 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.

6.2.5 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.

6.2.6 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)
```

6.2.7 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)
```

Chapter 7

Perspective Broker

7.1 Overview of Twisted Spread

Perspective Broker (affectionately known as “PB”) is an asynchronous, symmetric¹ network protocol for secure, remote method calls and transferring of objects. PB is “translucent, not transparent”, meaning that it is very visible and obvious to see the difference between local method calls and potentially remote method calls, but remote method calls are still extremely convenient to make, and it is easy to emulate them to have objects which work both locally and remotely.

PB supports user-defined serialized data in return values, which can be either copied each time the value is returned, or “cached”: only copied once and updated by notifications.

PB gets its name from the fact that access to objects is through a “perspective”. This means that when you are responding to a remote method call, you can establish who is making the call.

7.1.1 Rationale

No other currently existing protocols have all the properties of PB at the same time. The particularly interesting combination of attributes, though, is that PB is flexible and lightweight, allowing for rapid development, while still powerful enough to do two-way method calls and user-defined data types.

It is important to have these attributes in order to allow for a protocol which is extensible. One of the facets of this flexibility is that PB can integrate an arbitrary number of services could be aggregated over a single connection, as well as publish and call new methods on existing objects without restarting the server or client.

7.2 Introduction to Perspective Broker

7.2.1 Introduction

Suppose you find yourself in control of both ends of the wire: you have two programs that need to talk to each other, and you get to use any protocol you want. If you can think of your problem in terms of objects that need to make method calls on each other, then chances are good that you can use twisted’s Perspective Broker protocol rather than trying to shoehorn your needs into something like HTTP, or implementing yet another RPC mechanism².

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.

¹There 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.

²Most of Twisted is like this. Hell, most of unix is like this: if *you* think it would be useful, someone else has probably thought that way in the past, and acted on it, and you can take advantage of the tool they created to solve the same problem you’re facing now.

This document will contain several examples that will (hopefully) appear redundant and verbose once you’ve figured out what’s going on. To begin with, much of the code will just be labelled “magic”: don’t worry about how these parts work yet. It will be explained more fully later.

7.2.2 Object Roadmap

To start with, here are the major classes, interfaces, and functions involved in PB, with links to the file where they are defined (all of which are under `twisted/`, of course). Don’t worry about understanding what they all do yet: it’s easier to figure them out through their interaction than explaining them one at a time.

- *Factory*: `internet/protocol.py`
- *PBServerFactory*: `spread/pb.py`
- *Broker*: `spread/pb.py`

Other classes that are involved at some point:

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

Classes and interfaces that get involved when you start to care about authorization and security:

- *Portal*: `cred/portal.py`
- *IRealm*: `cred/portal.py`
- *IPerspective*: `spread/pb.py`, which you will usually be interacting with via *pb.Avatar* (a basic implementor of the interface).

Subclassing and Implementing

Technically you can subclass anything you want, but technically you could also write a whole new framework, which would just waste a lot of time. Knowing which classes are useful to subclass or which interfaces to implement is one of the bits of knowledge that’s crucial to using PB (and all of Twisted) successfully. Here are some hints to get started:

- *pb.Root*, *pb.Referenceable*: you’ll subclass these to make remotely-referenceable objects (i.e., objects which you can call methods on remotely) using PB. You don’t need to change any of the existing behavior, just inherit all of it and add the remotely-accessible methods that you want to export.
- *pb.Avatar*: You’ll be subclassing this when you get into PB programming with authorization. This is an implementor of *IPerspective*.
- *ICredentialsChecker*: Implement this if you want to authenticate your users against some sort of data store: i.e., an LDAP database, an RDBMS, etc. There are already a few implementations of this for various back-ends in `twisted.cred.checkers`.

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

7.2.3 Things you can Call Remotely

At this writing, there are three “flavors” of objects that can be accessed remotely through *RemoteReference* objects. Each of these flavors has a rule for how the `callRemote` message is transformed into a local method call on the server. In order to use one of these “flavors”, subclass them and name your published methods with the appropriate prefix.

- `twisted.spread.pb.IPerspective` implementors

This is the first interface we deal with. It is a “perspective” onto your PB application. Perspectives are slightly special because they are usually the first object that a given user can access in your application (after they log on). A user should only receive a reference to their *own* perspective. PB works hard to verify, as best it can, that any method that can be called on a perspective directly is being called on behalf of the user who is represented by that perspective. (Services with unusual requirements for “on behalf of”, such as simulations with the ability to possess another player’s avatar, are accomplished by providing indirect access to another user’s perspective.)

Perspectives are not usually serialized as remote references, so do not return an `IPerspective`-implementor directly.

The way most people will want to implement `IPerspective` is by subclassing `pb.Avatar`. Remotely accessible methods on `pb.Avatar` instances are named with the `perspective_` prefix.

- `twisted.spread.flavors.Referenceable`

Referenceable objects are the simplest kind of PB object. You can call methods on them and return them from methods to provide access to other objects’ methods.

However, when a method is called on a `Referenceable`, it’s not possible to tell who called it.

Remotely accessible methods on `Referenceables` are named with the `remote_` prefix.

- `twisted.spread.flavors.Viewable`

Viewable objects are remotely referenceable objects which have the additional requirement that it must be possible to tell who is calling them. The argument list to a `Viewable`’s remote methods is modified in order to include the `Perspective` representing the calling user.

Remotely accessible methods on `Viewables` are named with the `view_` prefix.

7.2.4 Things you can Copy Remotely

In addition to returning objects that you can call remote methods on, you can return structured copies of local objects.

There are 2 basic flavors that allow for copying objects remotely. Again, you can use these by subclassing them. In order to specify what state you want to have copied when these are serialized, you can either use the Python default `__getstate__` or specialized method calls for that flavor.

- `twisted.spread.flavors.Copyable`

This is the simpler kind of object that can be copied. Every time this object is returned from a method or passed as an argument, it is serialized and unserialized.

`Copyable` provides a method you can override, `getStateToCopyFor(perspective)`, which allows you to decide what an object will look like for the perspective who is requesting it. The `perspective` argument will be the perspective which is either passing an argument or returning a result an instance of your `Copyable` class.

For security reasons, in order to allow a particular `Copyable` class to actually be copied, you must declare a `RemoteCopy` handler for that `Copyable` subclass. The easiest way to do this is to declare both in the same module, like so:

```
from twisted.spread import flavors
class Foo(flavors.Copyable):
    pass
class RemoteFoo(flavors.RemoteCopy):
    pass
flavors.setCopierForClass(str(Foo), RemoteFoo)
```

In this case, each time a `Foo` is copied between peers, a `RemoteFoo` will be instantiated and populated with the `Foo`’s state. If you do not do this, PB will complain that there have been security violations, and it may close the connection.

- `twisted.spread.flavors.Cacheable`

Let me preface this with a warning: `Cacheable` may be hard to understand. The motivation for it may be unclear if you don't have some experience with real-world applications that use remote method calling of some kind. Once you understand why you need it, what it does will likely seem simple and obvious, but if you get confused by this, forget about it and come back later. It's possible to use PB without understanding `Cacheable` at all.

`Cacheable` is a flavor which is designed to be copied only when necessary, and updated on the fly as changes are made to it. When passed as an argument or a return value, if a `Cacheable` exists on the side of the connection it is being copied to, it will be referred to by ID and not copied.

`Cacheable` is designed to minimize errors involved in replicating an object between multiple servers, especially those related to having stale information. In order to do this, `Cacheable` automatically registers observers and queries state atomically, together. You can override the method `getStateToCacheAndObserveFor(self, perspective, observer)` in order to specify how your observers will be stored and updated.

Similar to `getStateToCopyFor`, `getStateToCacheAndObserveFor` gets passed a perspective. It also gets passed an observer, which is a remote reference to a "secret" fourth referenceable flavor: `RemoteCache`.

A `RemoteCache` is simply the object that represents your `Cacheable` on the other side of the connection. It is registered using the same method as `RemoteCopy`, above. `RemoteCache` is different, however, in that it will be referenced by its peer. It acts as a `Referenceable`, where all methods prefixed with `observe_` will be callable remotely. It is recommended that your object maintain a list (note: library support for this is forthcoming!) of observers, and update them using `callRemote` when the `Cacheable` changes in a way that should be noticeable to its clients.

Finally, when all references to a `Cacheable` from a given perspective are lost, `stoppedObserving(perspective, observer)` will be called on the `Cacheable`, with the same perspective/observer pair that `getStateToCacheAndObserveFor` was originally called with. Any cleanup remote calls can be made there, as well as removing the observer object from any lists which it was previously in. Any further calls to this observer object will be invalid.

7.3 Using Perspective Broker

7.3.1 Basic Example

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

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

class Echoer(pb.Root):
    def remote_echo(self, st):
        print 'echoing:', st
        return st

if __name__ == '__main__':
    reactor.listenTCP(8789, pb.PBServerFactory(Echoer()))
    reactor.run()
```

Source listing — *pbsimple.py*

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

factory = pb.PBClientFactory()
```

```

reactor.connectTCP("localhost", 8789, factory)
d = factory.getRootObject()
d.addCallback(lambda object: object.callRemote("echo", "hello network"))
d.addCallback(lambda echo: 'server echoed: '+echo)
d.addErrback(lambda reason: 'error: '+str(reason.value))
d.addCallback(util.println)
d.addCallback(lambda _: reactor.stop())
reactor.run()

```

Source listing — *pbsimpleclient.py*

First we look at the server. This defines an `Echoer` class (derived from `pb.Root`), with a method called `remote_echo()`. `pb.Root` objects (because of their inheritance of `pb.Referenceable`, described later) can define methods with names of the form `remote_*`; a client which obtains a remote reference to that `pb.Root` object will be able to invoke those methods.

The `pb.Root`-ish object is given to a `pb.PBServerFactory()`. This is a `Factory` object like any other: the `Protocol` objects it creates for new connections know how to speak the PB protocol. The object you give to `pb.PBServerFactory()` becomes the “root object”, which simply makes it available for the client to retrieve. The client may only request references to the objects you want to provide it: this helps you implement your security model. Because it is so common to export just a single object (and because a `remote_*` method on that one can return a reference to any other object you might want to give out), the simplest example is one where the `PBServerFactory` is given the root object, and the client retrieves it.

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

Because `.getRootObject()` has to wait until a network connection has been made and exchange some data, it may take a while, so it returns a `Deferred`, to which the `gotObject()` callback is attached. (See the documentation on *Deferring Execution* (page 99) 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 `Deferred`s turns out to be a very clean way to deal with the whole thing.

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

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

method calls³. If you have an object that is `Referenceable`, then any client that manages to get a reference to it can invoke any `remote_*` methods they please.

Note:

The *only* thing they can do is invoke those methods. In particular, they cannot access attributes. From a security point of view, you control what they can do by limiting what the `remote_*` methods can do.

Also note: the other classes like `Referenceable` allow access to other methods, in particular `perspective_*` and `view_*` may be accessed. Don't write local-only methods with these names, because then remote callers will be able to do more than you intended.

Also also note: the other classes like `pb.Copyable` do allow access to attributes, but you control which ones they can see.

You don't have to be a `pb.Root` to be remotely callable, but you do have to be `pb.Referenceable`. (Objects that inherit from `pb.Referenceable` but not from `pb.Root` can be remotely called, but only `pb.Root`-ish objects can be given to the `PBServerFactory`.)

7.3.2 Complete Example

```
from twisted.spread import pb

class QuoteReader(pb.Root):

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

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

QuoteReader Root object — *pbquote.py*

For examples of these, we're returning to the `TwistedQuotes` project discussed in *Writing Plugins* (page 140). To use the examples in this HOWTO, we need to make a TML file to refer to our new set of examples:

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

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

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

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

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

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

from twisted.application import service, internet
```

³There are a few other classes that can bestow this ability, but `pb.Referenceable` is the easiest to understand; see 'flavors' below for details on the others.

```

from twisted.python import usage          # twisted command-line processing
from twisted.spread import pb            # Perspective Broker

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

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

```

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

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

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

```

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

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

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

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

factory = pb.PBClientFactory()
reactor.connectTCP(

```

```

    "localhost", # host name
    pb.portno, # port number
    factory, # factory
)

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

reactor.run() # start the main loop

```

PB Quotes Client Code — *pbquoteclient.py*

`pb.PBClientFactory.getRootObject` will handle all the details of waiting for the creation of a connection. It returns a `Deferred`, which will have its callback called when the reactor connects to the remote server and `pb.PBClientFactory` gets the root, and have its errback called when the object-connection fails for any reason, whether it was host lookup failure, connection refusal, or some server-side error.

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

```

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

```

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

`RemoteReference` objects have one method which is their purpose for being: `callRemote`. This method allows you to call a remote method on the object being referred to by the `Reference`. `RemoteReference.callRemote`, like `pb.PBClientFactory.getRootObject`, returns a `Deferred`. When a response to the method-call being sent arrives, the `Deferred`'s callback or errback will be made, depending on whether an error occurred in processing the method call.

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

7.3.3 Passing more references

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

```

#!/usr/bin/python
from twisted.spread import pb

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

class One(pb.Root):
    def remote_getTwo(self):
        two = Two()
        print "returning a Two called", two
        return two

from twisted.internet import reactor
reactor.listenTCP(8800, pb.PBServerFactory(One()))
reactor.run()

```

Source listing — *pb1server.py*

```

#!/usr/bin/python

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

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

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

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

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

main()

```

Source listing — *pb1client.py*

The root object has a method called `remote_getTwo`, which returns the `Two()` instance. On the client end, the callback gets a `RemoteReference` to that instance. The client can then invoke two's `.remote_three()` method.

You can use this technique to provide access to arbitrary sets of objects. Just remember that any object that might get passed “over the wire” must inherit from `Referenceable` (or one of the other flavors). If you try to pass a non-`Referenceable` object (say, by returning one from a `remote_*` method), you'll get an `InsecureJelly` exception⁴.

7.3.4 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()`⁵.

```

#!/usr/bin/python

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

```

⁴This can be overridden, by subclassing one of the `Serializable` flavors and defining custom serialization code for your class. See *Passing Complex Types* (page 189) for details.

⁵The 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.

```

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

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

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

```

Source listing — *pb2server.py*

```

#!/usr/bin/python

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

def main():
    foo = Foo()
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 8800, factory)
    factory.getRootObject().addCallback(foo.step1)
    reactor.run()

# keeping globals around is starting to get ugly, so we use a simple class
# instead. Instead of hooking one function to the next, we hook one method
# to the next.

class Foo:
    def __init__(self):
        self.oneRef = None

    def step1(self, obj):
        print "got one object:", obj
        self.oneRef = obj
        print "asking it to getTwo"
        self.oneRef.callRemote("getTwo").addCallback(self.step2)

    def step2(self, two):
        print "got two object:", two
        print "giving it back to one"
        print "one is", self.oneRef
        self.oneRef.callRemote("checkTwo", two)

```

```
main()
```

Source listing — *pb2client.py*

The server gives a `Two()` instance to the client, who then returns the reference back to the server. The server compares the “two” given with the “two” received and shows that they are the same, and that both are real objects instead of remote references.

A few other techniques are demonstrated in `pb2client.py`. One is that the callbacks are added with `.addCallback` instead of `.addCallbacks`. As you can tell from the *Deferred* (page 99) 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.

7.3.5 References to client-side objects

Anything that’s `Referenceable` can get passed across the wire, *in either direction*. The “client” can give a reference to the “server”, and then the server can use `.callRemote()` to invoke methods on the client end. This fuzzes the distinction between “client” and “server”: the only real difference is who initiates the original TCP connection; after that it’s all symmetric.

```
#!/usr/bin/python

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

class One(pb.Root):
    def remote_takeTwo(self, two):
        print "received a Two called", two
        print "telling it to print(12)"
        two.callRemote("print", 12)

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

Source listing — *pb3server.py*

```
#!/usr/bin/python

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

class Two(pb.Referenceable):
    def remote_print(self, arg):
        print "Two.print() called with", arg

def main():
    two = Two()
```



```

factory = pb.PBClientFactory()
reactor.connectTCP("localhost", 8800, factory)
defl = factory.getRootObject()
defl.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.

7.3.6 Raising Remote Exceptions

Everything so far has covered what happens when things go right. What about when they go wrong? The Python Way is to raise an exception of some sort. The Twisted Way is the same.

The only special thing you do is to define your `Exception` subclass by deriving it from `pb.Error`. When any remotely-invokable method (like `remote_*` or `perspective_*`) raises a `pb.Error`-derived exception, a serialized form of that `Exception` object will be sent back over the wire⁶. 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
from twisted.internet import reactor

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

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

class One(pb.Root):
    def remote_broken(self):
        msg = "fall down go boom"
        print "raising a MyError exception with data '%s'" % msg
        raise MyError(msg)
    def remote_broken2(self):

```

⁶To 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`.

```

        msg = "hadda owie"
        print "raising a MyError2 exception with data '%s'" % msg
        raise MyError2(msg)

def main():
    reactor.listenTCP(8800, pb.PBServerFactory(One()))
    reactor.run()

if __name__ == '__main__':
    main()

```

Source listing — *exc_server.py*

```

#!/usr/bin/python

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

def main():
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 8800, factory)
    d = factory.getRootObject()
    d.addCallbacks(got_obj)
    reactor.run()

def got_obj(obj):
    # change "broken" into "broken2" to demonstrate an unhandled exception
    d2 = obj.callRemote("broken")
    d2.addCallback(working)
    d2.addErrback(broken)

def working():
    print "erm, it wasn't *supposed* to work.."

def broken(reason):
    print "got remote Exception"
    # reason should be a Failure (or subclass) holding the MyError exception
    print ".__class__ =", reason.__class__
    print ".getErrorMessage() =", reason.getErrorMessage()
    print ".type =", reason.type
    reactor.stop()

main()

```

Source listing — *exc_client.py*

```

% ./exc_client.py
got remote Exception
.__class__ = twisted.spread.pb.CopiedFailure
.getErrorMessage() = fall down go boom
.type = __main__.MyError
Main loop terminated.

```

Oh, and what happens if you raise some other kind of exception? Something that *isn't* subclassed from `pb.Error`? Well, those are called “unexpected exceptions”, which make Twisted think that something has *really* gone wrong. These will raise an exception on the *server* side. This won't break the connection (the exception is trapped,

just like most exceptions that occur in response to network traffic), but it will print out an unsightly stack trace on the server's stderr with a message that says "Peer Will Receive PB Traceback", just as if the exception had happened outside a remotely-invokable method. (This message will go the current log target, if `log.startLogging` was used to redirect it). The client will get the same `Failure` object in either case, but subclassing your exception from `pb.Error` is the way to tell Twisted that you expect this sort of exception, and that it is ok to just let the client handle it instead of also asking the server to complain. Look at `exc_client.py` and change it to invoke `broken2()` instead of `broken()` to see the change in the server's behavior.

If you don't add an `errback` function to the `Deferred`, then a remote exception will still send a `Failure` object back over, but it will get lodged in the `Deferred` with nowhere to go. When that `Deferred` finally goes out of scope, the side that did `callRemote` will emit a message about an "Unhandled error in Deferred", along with an ugly stack trace. It can't raise an exception at that point (after all, the `callRemote` that triggered the problem is long gone), but it will emit a traceback. So be a good programmer and *always add errback handlers*, even if they are just calls to `log.err`.

7.3.7 Try/Except blocks and `Failure.trap`

To implement the equivalent of the Python try/except blocks (which can trap particular kinds of exceptions and pass others "up" to higher-level try/except blocks), you can use the `.trap()` method in conjunction with multiple `errback` handlers on the `Deferred`. Re-raising an exception in an `errback` handler serves to pass that new exception to the next handler in the chain. The `trap` method is given a list of exceptions to look for, and will re-raise anything that isn't on the list. Instead of passing unhandled exceptions "up" to an enclosing try block, this has the effect of passing the exception "off" to later `errback` handlers on the same `Deferred`. The `trap` calls are used in chained `errbacks` to test for each kind of exception in sequence.

```
#!/usr/bin/python

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

class MyException(pb.Error):
    pass

class One(pb.Root):
    def remote_fooMethod(self, arg):
        if arg == "panic!":
            raise MyException
        return "response"
    def remote_shutdown(self):
        reactor.stop()

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

Source listing — *trap_server.py*

```
#!/usr/bin/python

from twisted.spread import pb, jelly
from twisted.python import log
from twisted.internet import reactor

class MyException(pb.Error): pass
class MyOtherException(pb.Error): pass

class ScaryObject:
    # not safe for serialization
    pass
```

```

def worksLike(obj):
    # the callback/errback sequence in class One works just like an
    # asynchronous version of the following:
    try:
        response = obj.callMethod(name, arg)
    except pb.DeadReferenceError:
        print " stale reference: the client disconnected or crashed"
    except jelly.InsecureJelly:
        print " InsecureJelly: you tried to send something unsafe to them"
    except (MyException, MyOtherException):
        print " remote raised a MyException" # or MyOtherException
    except:
        print " something else happened"
    else:
        print " method successful, response:", response

class One:
    def worked(self, response):
        print " method successful, response:", response
    def check_InsecureJelly(self, failure):
        failure.trap(jelly.InsecureJelly)
        print " InsecureJelly: you tried to send something unsafe to them"
        return None
    def check_MyException(self, failure):
        which = failure.trap(MyException, MyOtherException)
        if which == MyException:
            print " remote raised a MyException"
        else:
            print " remote raised a MyOtherException"
        return None
    def catch_everythingElse(self, failure):
        print " something else happened"
        log.err(failure)
        return None

    def doCall(self, explanation, arg):
        print explanation
        try:
            deferred = self.remote.callRemote("fooMethod", arg)
            deferred.addCallback(self.worked)
            deferred.addErrback(self.check_InsecureJelly)
            deferred.addErrback(self.check_MyException)
            deferred.addErrback(self.catch_everythingElse)
        except pb.DeadReferenceError:
            print " stale reference: the client disconnected or crashed"

    def callOne(self):
        self.doCall("callOne: call with safe object", "safe string")
    def callTwo(self):
        self.doCall("callTwo: call with dangerous object", ScaryObject())
    def callThree(self):
        self.doCall("callThree: call that raises remote exception", "panic!")
    def callShutdown(self):
        print "telling them to shut down"
        self.remote.callRemote("shutdown")
    def callFour(self):

```

```

self.doCall("callFour: call on stale reference", "dummy")

def got_obj(self, obj):
    self.remote = obj
    reactor.callLater(1, self.callOne)
    reactor.callLater(2, self.callTwo)
    reactor.callLater(3, self.callThree)
    reactor.callLater(4, self.callShutdown)
    reactor.callLater(5, self.callFour)
    reactor.callLater(6, reactor.stop)

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

```

Source listing — *trap_client.py*

```

% ./trap_client.py
callOne: call with safe object
  method successful, response: response
callTwo: call with dangerous object
  InsecureJelly: you tried to send something unsafe to them
callThree: call that raises remote exception
  remote raised a MyException
telling them to shut down
callFour: call on stale reference
  stale reference: the client disconnected or crashed
%

```

In this example, `callTwo` tries to send an instance of a locally-defined class through `callRemote`. The default security model implemented by `pb.Jelly` on the remote end will not allow unknown classes to be unserialized (i.e. taken off the wire as a stream of bytes and turned back into an object: a living, breathing instance of some class): one reason is that it does not know which local class ought to be used to create an instance that corresponds to the remote object⁷. 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`.

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

⁷The 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 *Passing Complex Types* (page 189) for an example.

that point may only cause a log message (which could even be thrown away if `log.startLogging` is not used to point it at `stdout` or a log file). In contrast, a synchronous exception that is not handled by any other `except: block` will very visibly terminate the program immediately with a noisy stack trace.

`callFour` shows another kind of exception that can occur while using `callRemote: pb.DeadReferenceError`. This one occurs when the remote end has disconnected or crashed, leaving the local side with a stale reference. This kind of exception happens to be reported right away (XXX: is this guaranteed? probably not), so must be caught in a traditional synchronous `try: except pb.DeadReferenceError block`.

Yet another kind that can occur is a `pb.PBConnectionLost` exception. This occurs (asynchronously) if the connection was lost while you were waiting for a `callRemote` call to complete. When the line goes dead, all pending requests are terminated with this exception. Note that you have no way of knowing whether the request made it to the other end or not, nor how far along in processing it they had managed before the connection was lost. XXX: explain transaction semantics, find a decent reference.

7.4 PB Copyable: Passing Complex Types

7.4.1 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`).

7.4.2 Motivation

From the *previous chapter* (page 176), 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.

7.4.3 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 be used later to retrieve a document. Then

imagine what if that object were created with `os.fopen("~/gnupg/secring.gpg")`. Or an instance of `telnetlib.Telnet("localhost", "chargen")`.

Classes you’ve written for your own program are likely to have far more power. They may run code during `__init__`, or even have special meaning simply because of their existence. A program might have `User` objects to represent user accounts, and have a rule that says all `User` objects in the system are referenced when authorizing a login session. (In this system, `User.__init__` would probably add the object to a global list of known users). The simple act of creating an object would give access to somebody. If you could be tricked into creating a bad object, an unauthorized user would get access.

So object creation needs to be part of a system’s security design. The dotted line between “trusted inside” and “untrusted outside” needs to describe what may be done in response to outside events. One of those events is the receipt of an object through a PB remote procedure call, which is a request to create an object in your “inside” namespace. The question is what to do in response to it. For this reason, you must explicitly specify what remote classes will be accepted, and how their local representatives are to be created.

What class to use?

Another basic question to answer before we can do anything useful with an incoming serialized object is: what class should we create? The simplistic answer is to create the “same kind” that was serialized on the sender’s end of the wire, but this is not as easy or as straightforward as you might think. Remember that the request is coming from a different program, using a potentially different set of class libraries. In fact, since PB has also been implemented in Java, Emacs-Lisp, and other languages, there’s no guarantee that the sender is even running Python! All we know on the receiving end is a list of two things which describe the instance they are trying to send us: the name of the class, and a representation of the contents of the object.

PB lets you specify the mapping from remote class names to local classes with the `setUnjellyableForClass` function⁸. This function takes a remote/sender class reference (either the fully-qualified name as used by the sending end, or a class object from which the name can be extracted), and a local/recipient class (used to create the local representation for incoming serialized objects). Whenever the remote end sends an object, the class name that they transmit is looked up in the table controlled by this function. If a matching class is found, it is used to create the local object. If not, you get the `InsecureJelly` exception.

In general you expect both ends to share the same codebase: either you control the program that is running on both ends of the wire, or both programs share some kind of common language that is implemented in code which exists on both ends. You wouldn’t expect them to send you an object of the `MyFooziWhatZit` class unless you also had a definition for that class. So it is reasonable for the Jelly layer to reject all incoming classes except the ones that you have explicitly marked with `setUnjellyableForClass`. But keep in mind that the sender’s idea of a `User` object might differ from the recipient’s, either through namespace collisions between unrelated packages, version skew between nodes that haven’t been updated at the same rate, or a malicious intruder trying to cause your code to fail in some interesting or potentially vulnerable way.

7.4.4 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
```

⁸Note that, in this context, “unjelly” is a verb with the opposite meaning of “jelly”. The verb “to jelly” means to serialize an object or data structure into a sequence of bytes (or other primitive transmittable/storable representation), while “to unjelly” means to unserialize the bytestream into a live object in the receiver’s memory space. “Unjellyable” is a noun, (*not* an adjective), referring to the class that serves as a destination or recipient of the unjellying process. “A is unjellyable into B” means that a serialized representation A (of some remote object) can be unserialized into a local object of type B. It is these objects “B” that are the “Unjellyable” second argument of the `setUnjellyableForClass` function.

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

```

        self.numFrogs = numFrogs
    def countFrogs(self):
        print "%d frogs" % self.numFrogs

class CopyPond(LilyPond, pb.Copyable):
    pass

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

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

    def ok(self, response):
        print "pond arrived", response
        reactor.stop()

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

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

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

if __name__ == '__main__':
    main()

```

Source listing — *copy_sender.py*

"""PB copy receiver example.

This is a Twisted Application Configuration (tac) file. Run with e.g.

twistd -ny copy_receiver.tac

See the twistd(1) man page or

<http://twistedmatrix.com/documents/current/howto/application> for details.

"""


```

import sys
if __name__ == '__main__':
    print __doc__
    sys.exit(1)

from twisted.application import service, internet
from twisted.internet import reactor
from twisted.spread import pb
from copy_sender import LilyPond, CopyPond

from twisted.python import log
#log.startLogging(sys.stdout)

class ReceiverPond(pb.RemoteCopy, LilyPond):
    pass
pb.setUnjellyableForClass(CopyPond, ReceiverPond)

class Receiver(pb.Root):
    def remote_takePond(self, pond):
        print " got pond:", pond
        pond.countFrogs()
        return "safe and sound" # positive acknowledgement
    def remote_shutdown(self):
        reactor.stop()

application = service.Application("copy_receiver")
internet.TCPServer(8800, pb.PBServerFactory(Receiver())).setServiceParent(
    service.IServiceCollection(application))

```

Source listing — *copy_receiver.tac*

The sending side has a class called `LilyPond`. To make this eligible for transport through `callRemote` (either as an argument, a return value, or something referenced by either of those [like a dictionary value]), it must inherit from one of the four `Serializable` classes. In this section, we focus on `Copyable`. The copyable subclass of `LilyPond` is called `CopyPond`. We create an instance of it and send it through `callRemote` as an argument to the receiver's `remote_takePond` method. The Jelly layer will serialize (“jelly”) that object as an instance with a class name of “`copy_sender.CopyPond`” and some chunk of data that represents the object's state. `pond.__class__.__module__` and `pond.__class__.__name__` are used to derive the class name string. The object's `getStateToCopy` method is used to get the state: this is provided by `pb.Copyable`, and the default just retrieves `self.__dict__`. This works just like the optional `__getstate__` method used by `pickle`. The pair of name and state are sent over the wire to the receiver.

The receiving end defines a local class named `ReceiverPond` to represent incoming `LilyPond` instances. This class derives from the sender's `LilyPond` class (with a fully-qualified name of `copy_sender.LilyPond`), which specifies how we expect it to behave. We trust that this is the same `LilyPond` class as the sender used. (At the very least, we hope ours will be able to accept a state created by theirs). It also inherits from `pb.RemoteCopy`, which is a requirement for all classes that act in this local-representative role (those which are given to the second argument of `setUnjellyableForClass`). `RemoteCopy` provides the methods that tell the Jelly layer how to create the local object from the incoming serialized state.

Then `setUnjellyableForClass` is used to register the two classes. This has two effects: instances of the remote class (the first argument) will be allowed in through the security layer, and instances of the local class (the second argument) will be used to contain the state that is transmitted when the sender serializes the remote object.

When the receiver unserializes (“unjellies”) the object, it will create an instance of the local `ReceiverPond` class, and hand the transmitted state (usually in the form of a dictionary) to that object's `setCopyableState` method. This acts just like the `__setstate__` method that `pickle` uses when unserializing an object. `getStateToCopy`/`setCopyableState` are distinct from `__getstate__`/`__setstate__` to allow objects to be persisted (across time) differently than they are transmitted (across [memory]space).

When this is run, it produces the following output:

```
[ - ] twisted.spread.pb.PBServerFactory starting on 8800
[ - ] Starting factory <twisted.spread.pb.PBServerFactory instance at
0x406159cc>
[Broker,0,127.0.0.1] got pond: <__builtin__.ReceiverPond instance at
0x406ec5ec>
[Broker,0,127.0.0.1] 7 frogs

% ./copy_sender.py
7 frogs
copy_sender.CopyPond
pond arrived safe and sound
Main loop terminated.
%
```

Controlling the Copied State

By overriding `getStateToCopy` and `setCopyableState`, you can control how the object is transmitted over the wire. For example, you might want perform some data-reduction: pre-compute some results instead of sending all the raw data over the wire. Or you could replace references to a local object on the sender's side with markers before sending, then upon receipt replace those markers with references to a receiver-side proxy that could perform the same operations against a local cache of data.

Another good use for `getStateToCopy` is to implement “local-only” attributes: data that is only accessible by the local process, not to any remote users. For example, a `.password` attribute could be removed from the object state before sending to a remote system. Combined with the fact that `Copyable` objects return unchanged from a round trip, this could be used to build a challenge-response system (in fact PB does this with `pb.Referenceable` objects to implement authorization as described *here* (page 200)).

Whatever `getStateToCopy` returns from the sending object will be serialized and sent over the wire; `setCopyableState` gets whatever comes over the wire and is responsible for setting up the state of the object it lives in.

```
#!/usr/bin/python

from twisted.spread import pb

class FrogPond:
    def __init__(self, numFrogs, numToads):
        self.numFrogs = numFrogs
        self.numToads = numToads
    def count(self):
        return self.numFrogs + self.numToads

class SenderPond(FrogPond, pb.Copyable):
    def getStateToCopy(self):
        d = self.__dict__.copy()
        d['frogsAndToads'] = d['numFrogs'] + d['numToads']
        del d['numFrogs']
        del d['numToads']
        return d

class ReceiverPond(pb.RemoteCopy):
    def setCopyableState(self, state):
        self.__dict__ = state
    def count(self):
        return self.frogsAndToads

pb.setUnjellyableForClass(SenderPond, ReceiverPond)
```

Source listing — *copy2_classes.py*

```

#!/usr/bin/python

from twisted.spread import pb, jelly
from twisted.python import log
from twisted.internet import reactor
from copy2_classes import SenderPond

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

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

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

def main():
    pond = SenderPond(3, 4)
    print "count %d" % pond.count()

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

if __name__ == '__main__':
    main()

```

Source listing — *copy2_sender.py*

```

#!/usr/bin/python

from twisted.application import service, internet
from twisted.internet import reactor
from twisted.spread import pb
import copy2_classes # needed to get ReceiverPond registered with Jelly

class Receiver(pb.Root):
    def remote_takePond(self, pond):
        print " got pond:", pond

```

```

        print " count %d" % pond.count()
        return "safe and sound" # positive acknowledgement
    def remote_shutdown(self):
        reactor.stop()

application = service.Application("copy_receiver")
internet.TCPServer(8800, pb.PBServerFactory(Receiver())).setServiceParent(
    service.IServiceCollection(application))

```

Source listing — *copy2_receiver.py*

In this example, the classes are defined in a separate source file, which also sets up the binding between them. The `SenderPond` and `ReceiverPond` are unrelated save for this binding: they happen to implement the same methods, but use different internal instance variables to accomplish them.

The recipient of the object doesn't even have to import the class definition into their namespace. It is sufficient that they import the class definition (and thus execute the `setUnjellyableForClass` statement). The Jelly layer remembers the class definition until a matching object is received. The sender of the object needs the definition, of course, to create the object in the first place.

When run, the `copy2` example emits the following:

```

% twisted -n -y copy2_receiver.py
[-] twisted.spread.pb.PBServerFactory starting on 8800
[-] Starting factory <twisted.spread.pb.PBServerFactory instance at
0x40604b4c>
[Broker,0,127.0.0.1] got pond: <copy2_classes.ReceiverPond instance at
0x406eb2ac>
[Broker,0,127.0.0.1] count 7

% ./copy2_sender.py
count 7
pond arrived safe and sound
Main loop terminated.
%

```

Things To Watch Out For

- The first argument to `setUnjellyableForClass` must refer to the class *as known by the sender*. The sender has no way of knowing about how your local `import` statements are set up, and Python's flexible namespace semantics allow you to access the same class through a variety of different names. You must match whatever the sender does. Having both ends import the class from a separate file, using a canonical module name (no "sibling imports"), is a good way to get this right, especially when both the sending and the receiving classes are defined together, with the `setUnjellyableForClass` immediately following them. (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`⁹.
- The same class can be used to send and receive. Just have it inherit from both `pb.Copyable` and `pb.RemoteCopy`. This will also make it possible to send the same class symmetrically back and forth over the wire. But don't get confused about when it is coming (and using `setCopyableState`) versus when it is going (using `getStateToCopy`).
- `InsecureJelly` exceptions are raised by the receiving end. They will be delivered asynchronously to an `errback` handler. If you do not add one to the `Deferred` returned by `callRemote`, then you will never receive notification of the problem.

⁹`pb.RemoteCopy` is actually defined as `flavors.RemoteCopy`, but `pb.RemoteCopy` is the preferred way to access it

- 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.

7.4.5 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 changed¹⁰.

You derive your sender-side class from `pb.Cacheable`, and you add two methods: `getStateToCacheAndObserveFor` and `stoppedObserving`. The first is called when a remote caching reference is first created, and retrieves the data with which the cache is first filled. It also provides an object called the “observer”¹¹ that points at that receiver-side cache. Every time the state of the object is changed, you give a message to the observer, informing them of the change. The other method, `stoppedObserving`, is called when the remote cache goes away, so that you can stop sending updates.

On the receiver end, you make your cache class inherit from `pb.RemoteCache`, and implement the `setCopyableState` as you would for a `pb.RemoteCopy` object. In addition, you must implement methods to receive the updates sent to the observer by the `pb.Cacheable`: these methods should have names that start with `observe_`, and match the `callRemote` invocations from the sender side just as the usual `remote_*` and `perspective_*` methods match normal `callRemote` calls.

The first time a reference to the `pb.Cacheable` object is sent to any particular recipient, a sender-side Observer will be created for it, and the `getStateToCacheAndObserveFor` method will be called to get the current state and register the Observer. The state which that returns is sent to the remote end and turned into a local representation using `setCopyableState` just like `pb.RemoteCopy`, described above (in fact it inherits from that class).

After that, your “setter” functions on the sender side should call `callRemote` on the Observer, which causes `observe_*` methods to run on the receiver, which are then supposed to update the receiver-local (cached) state.

When the receiver stops following the cached object and the last reference goes away, the `pb.RemoteCache` object can be freed. Just before it dies, it tells the sender side it no longer cares about the original object. When *that* reference count goes to zero, the Observer goes away and the `pb.Cacheable` object can stop announcing every change that takes place. The `stoppedObserving` method is used to tell the `pb.Cacheable` that the Observer has gone away.

With the `pb.Cacheable` and `pb.RemoteCache` classes in place, bound together by a call to `pb.setUnjellyableForClass`, all that remains is to pass a reference to your `pb.Cacheable` over the wire to the

¹⁰Of course you could be clever and add a hook to `__setattr__`, along with magical change-announcing subclasses of the usual builtin types, to detect changes that result from normal “=” set operations. The semi-magical “property attributes” that were introduced in Python-2.2 could be useful too. The result might be hard to maintain or extend, though.

¹¹this 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.

remote end. The corresponding `pb.RemoteCache` object will automatically be created, and the matching methods will be used to keep the receiver-side slave object in sync with the sender-side master object.

Example

Here is a complete example, in which the `MasterDuckPond` is controlled by the sending side, and the `SlaveDuckPond` is a cache that tracks changes to the master:

```
#!/usr/bin/python

from twisted.spread import pb

class MasterDuckPond(pb.Cacheable):
    def __init__(self, ducks):
        self.observers = []
        self.ducks = ducks
    def count(self):
        print "I have [%d] ducks" % len(self.ducks)
    def addDuck(self, duck):
        self.ducks.append(duck)
        for o in self.observers: o.callRemote('addDuck', duck)
    def removeDuck(self, duck):
        self.ducks.remove(duck)
        for o in self.observers: o.callRemote('removeDuck', duck)
    def getStateToCacheAndObserveFor(self, perspective, observer):
        self.observers.append(observer)
        # you should ignore pb.Cacheable-specific state, like self.observers
        return self.ducks # in this case, just a list of ducks
    def stoppedObserving(self, perspective, observer):
        self.observers.remove(observer)

class SlaveDuckPond(pb.RemoteCache):
    # This is a cache of a remote MasterDuckPond
    def count(self):
        return len(self.cacheducks)
    def getDucks(self):
        return self.cacheducks
    def setCopyableState(self, state):
        print "cache - sitting, er, setting ducks"
        self.cacheducks = state
    def observe_addDuck(self, newDuck):
        print "cache - addDuck"
        self.cacheducks.append(newDuck)
    def observe_removeDuck(self, deadDuck):
        print "cache - removeDuck"
        self.cacheducks.remove(deadDuck)

pb.setUnjellyableForClass(MasterDuckPond, SlaveDuckPond)
```

Source listing — *cache_classes.py*

```
#!/usr/bin/python

from twisted.spread import pb, jelly
from twisted.python import log
from twisted.internet import reactor
from cache_classes import MasterDuckPond
```

```

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

    def phase1(self, remote):
        self.remote = remote
        d = remote.callRemote("takePond", self.pond)
        d.addCallback(self.phase2).addErrback(log.err)
    def phase2(self, response):
        self.pond.addDuck("ugly duckling")
        self.pond.count()
        reactor.callLater(1, self.phase3)
    def phase3(self):
        d = self.remote.callRemote("checkDucks")
        d.addCallback(self.phase4).addErrback(log.err)
    def phase4(self, dummy):
        self.pond.removeDuck("one duck")
        self.pond.count()
        self.remote.callRemote("checkDucks")
        d = self.remote.callRemote("ignorePond")
        d.addCallback(self.phase5)
    def phase5(self, dummy):
        d = self.remote.callRemote("shutdown")
        d.addCallback(self.phase6)
    def phase6(self, dummy):
        reactor.stop()

def main():
    master = MasterDuckPond(["one duck", "two duck"])
    master.count()

    sender = Sender(master)
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 8800, factory)
    deferred = factory.getRootObject()
    deferred.addCallback(sender.phase1)
    reactor.run()

if __name__ == '__main__':
    main()

```

Source listing — *cache_sender.py*

```

#!/usr/bin/python

from twisted.application import service, internet
from twisted.internet import reactor
from twisted.spread import pb
import cache_classes

class Receiver(pb.Root):
    def remote_takePond(self, pond):
        self.pond = pond
        print "got pond:", pond # a DuckPondCache
        self.remote_checkDucks()

```

```

def remote_checkDucks(self):
    print "[%d] ducks: " % self.pond.count(), self.pond.getDucks()
def remote_ignorePond(self):
    # stop watching the pond
    print "dropping pond"
    # gc causes __del__ causes 'decache' msg causes stoppedObserving
    self.pond = None
def remote_shutdown(self):
    reactor.stop()

application = service.Application("copy_receiver")
internet.TCPServer(8800, pb.PBServerFactory(Receiver())).setServiceParent(
    service.IServiceCollection(application))

```

Source listing — *cache_receiver.py*

When run, this example emits the following:

```

% twisted -n -y cache_receiver.py
[-] twisted.spread.pb.PBServerFactory starting on 8800
[-] Starting factory <twisted.spread.pb.PBServerFactory instance at
0x40615acc>
[Broker,0,127.0.0.1] cache - sitting, er, setting ducks
[Broker,0,127.0.0.1] got pond: <cache_classes.SlaveDuckPond instance at
0x406eb5ec>
[Broker,0,127.0.0.1] [2] ducks: ['one duck', 'two duck']
[Broker,0,127.0.0.1] cache - addDuck
[Broker,0,127.0.0.1] [3] ducks: ['one duck', 'two duck', 'ugly duckling']
[Broker,0,127.0.0.1] cache - removeDuck
[Broker,0,127.0.0.1] [2] ducks: ['two duck', 'ugly duckling']
[Broker,0,127.0.0.1] dropping pond
%

% ./cache_sender.py
I have [2] ducks
I have [3] ducks
I have [2] ducks
Main loop terminated.
%

```

Points to notice:

- There is one Observer for each remote program that holds an active reference. Multiple references inside the same program don't matter: the serialization layer notices the duplicates and does the appropriate reference counting¹².
- Multiple Observers need to be kept in a list, and all of them need to be updated when something changes. By sending the initial state at the same time as you add the observer to the list, in a single atomic action that cannot be interrupted by a state change, you insure that you can send the same status update to all the observers.
- The `observer.callRemote` calls can still fail. If the remote side has disconnected very recently and `stoppedObserving` has not yet been called, you may get a `DeadReferenceError`. It is a good idea to add an `errback` to those `callRemotes` to throw away such an error. This is a useful idiom:

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

(XXX: verify that this is actually a concern)

¹²this 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.

- `getStateToCacheAndObserverFor` must return some object that represents the current state of the object. This may simply be the object's `__dict__` attribute. It is a good idea to remove the `pb.Cacheable`-specific members of it before sending it to the remote end. The list of Observers, in particular, should be left out, to avoid dizzying recursive `Cacheable` references. The mind boggles as to the potential consequences of leaving in such an item.
- A perspective argument is available to `getStateToCacheAndObserveFor`, as well as `stoppedObserving`. I think the purpose of this is to allow viewer-specific changes to the way the cache is updated. If all remote viewers are supposed to see the same data, it can be ignored.

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.

7.5 Authentication with Perspective Broker

7.5.1 Overview

The examples shown in *Using Perspective Broker* (page 176) demonstrate how to do basic remote method calls, but provided no facilities for authentication. In this context, authentication is about who gets which remote references, and how to restrict access to the “right” set of people or programs.

As soon as you have a program which offers services to multiple users, where those users should not be allowed to interfere with each other, you need to think about authentication. Many services use the idea of an “account”, and rely upon fact that each user has access to only one account. Twisted uses a system called *cred* (page 150) to handle authentication issues, and Perspective Broker has code to make it easy to implement the most common use cases.

7.5.2 Compartmentalizing Services

Imagine how you would write a chat server using PB. The first step might be a `ChatServer` object which had a bunch of `pb.RemoteReferences` that point at user clients. Pretend that those clients offered a `remote_print` method which lets the server print a message on the user's console. In that case, the server might look something like this:

[illegible]

For now, assume that all clients have somehow acquired a `pb.RemoteReference` to this `ChatServer` object, perhaps using `pb.Root` and `getRootObject` as described in the *previous chapter* (page 176). In this scheme, when a user sends a message to the group, their client runs something like the following:

```
remotegroup.callRemote("sendMessage", "alice", "Hi, my name is alice.")
```

Incorrect Arguments

You’ve probably seen the first problem: users can trivially spoof each other. We depend upon the user to pass a correct value in their “username” argument, and have no way to tell if they’re lying or not. There is nothing to prevent Alice from modifying her client to do:

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

much to the horror of Bob’s vegetarian friends.¹³

(In general, learn to get suspicious if you see any argument of a remotely-invokable method described as “must be X”)

The best way to fix this is to keep track of the user’s name locally, rather than asking them to send it to the server with each message. The best place to keep state is in an object, so this suggests we need a per-user object. Rather than choosing an obvious name¹⁴, let’s call this the `User` class.

```
class User(pb.Referenceable):
    def __init__(self, username, server, clientref):
        self.name = username
        self.server = server
        self.remote = clientref
    def remote_joinGroup(self, groupname):
        self.server.joinGroup(groupname, self)
    def remote_sendMessage(self, groupname, message):
        self.server.sendMessage(self.name, groupname, message)
    def send(self, message):
        self.remote.callRemote("print", message)

class ChatServer:
    def __init__(self):
        self.groups = {} # indexed by name
    def joinGroup(self, groupname, user):
        if not self.groups.has_key(groupname):
            self.groups[groupname] = []
        self.groups[groupname].append(user)
    def sendMessage(self, from_username, groupname, message):
        group = self.groups[groupname]
        if group:
            # send the message to all members of the group
            for user in group:
                user.send("<%s> says: %s" % (from_username, message))
```

Again, assume that each remote client gets access to a single `User` object, which is created with the proper username.

Note how the `ChatServer` object has no remote access: it isn’t even `pb.Referenceable` anymore. This means that all access to it must be mediated through other objects, with code that is under your control.

As long as Alice only has access to her own `User` object, she can no longer spoof Bob. The only way for her to invoke `ChatServer.sendMessage` is to call her `User` object’s `remote_sendMessage` method, and that method uses its own state to provide the `from_username` argument. It doesn’t give her any way to change that state.

¹³Apparently Alice is one of those weirdos who has nothing better to do than to try and impersonate Bob. She will lie to her chat client, send incorrect objects to remote methods, even rewrite her local client code entirely to accomplish this juvenile prank. Given this adversarial relationship, one must wonder why she and Bob seem to spend so much time together: their adventures are clearly documented by the cryptographic literature.

¹⁴the obvious name is clearly `ServerSidePerUserObjectWhichNobodyElseHasAccessTo`, but because python makes everything else so easy to read, it only seems fair to make your audience work for *something*

This restriction is important. The User object is able to maintain its own integrity because there is a wall between the object and the client: the client cannot inspect or modify internal state, like the `.name` attribute. The only way through this wall is via remote method invocations, and the only control Alice has over those invocations is when they get invoked and what arguments they are given.

Note:

No object can maintain its integrity against local threats: by design, Python offers no mechanism for class instances to hide their attributes, and once an intruder has a copy of `self.__dict__`, they can do everything the original object was able to do.

Unforgeable References

Now suppose you wanted to implement group parameters, for example a mode in which nobody was allowed to talk about mattresses because some users were sensitive and calming them down after someone said “mattress” is a hassle that were best avoided altogether. Again, per-group state implies a per-group object. We’ll go out on a limb and call this the Group object:

```
class User(pb.Referenceable):
    def __init__(self, username, server, clientref):
        self.name = username
        self.server = server
        self.remote = clientref
    def remote_joinGroup(self, groupname, allowMattress=True):
        return self.server.joinGroup(groupname, self)
    def send(self, message):
        self.remote.callRemote("print", message)

class Group(pb.Referenceable):
    def __init__(self, groupname, allowMattress):
        self.name = groupname
        self.allowMattress = allowMattress
        self.users = []
    def remote_send(self, from_user, message):
        if not self.allowMattress and message.find("mattress") != -1:
            raise ValueError, "Don't say that word"
        for user in self.users:
            user.send("<%s> says: %s" % (from_user.name, message))
    def addUser(self, user):
        self.users.append(user)

class ChatServer:
    def __init__(self):
        self.groups = {} # indexed by name
    def joinGroup(self, groupname, user, allowMattress):
        if not self.groups.has_key(groupname):
            self.groups[groupname] = Group(groupname, allowMattress)
        self.groups[groupname].addUser(user)
        return self.groups[groupname]
```

This example takes advantage of the fact that `pb.Referenceable` objects sent over a wire can be returned to you, and they will be turned into references to the same object that you originally sent. The client cannot modify the object in any way: all they can do is point at it and invoke its `remote_*` methods. Thus, you can be sure that the `.name` attribute remains the same as you left it. In this case, the client code would look something like this:

```
class ClientThing(pb.Referenceable):
    def remote_print(self, message):
        print message
    def join(self):
        d = self.remoteUser.callRemote("joinGroup", "#twisted",
```

```

allowMattress=False)
d.addCallback(self.gotGroup)
def gotGroup(self, group):
    group.callRemote("send", self.remoteUser, "hi everybody")

```

The `User` object is sent from the server side, and is turned into a `pb.RemoteReference` when it arrives at the client. The client sends it back to `Group.remote_send`, and PB turns it back into a reference to the original `User` when it gets there. `Group.remote_send` can then use its `.name` attribute as the sender of the message.

Note:

Third party references (there aren't any)

This technique also relies upon the fact that the `pb.Referenceable` reference can *only* come from someone who holds a corresponding `pb.RemoteReference`. The design of the serialization mechanism (implemented in `twisted.spread.jelly: pb, jelly, spread.. get it?` Also look for “banana” and “marmalade”. What other networking framework can claim API names based on sandwich ingredients?) makes it impossible for a client to obtain a reference that they weren't explicitly given. References passed over the wire are given id numbers and recorded in a per-connection dictionary. If you didn't give them the reference, the id number won't be in the dict, and no amount of guessing by a malicious client will give them anything else. The dict goes away when the connection is dropped, further limiting the scope of those references.

Futhermore, it is not possible for Bob to send *hisUser* reference to Alice (perhaps over some other PB channel just between the two of them). Outside the context of Bob's connection to the server, that reference is just a meaningless number. To prevent confusion, PB will tell you if you try to give it away: when you try to hand a `pb.RemoteReference` to a third party, you'll get an exception (implemented with an assert in `pb.py:364 RemoteReference.jellyFor`).

This helps the security model somewhat: only the client you gave the reference to can cause any damage with it. Of course, the client might be a brainless zombie, simply doing anything some third party wants. When it's not proxying `callRemote` invocations, it's probably terrorizing the living and searching out human brains for sustenance. In short, if you don't trust them, don't give them that reference.

And remember that everything you've ever given them over that connection can come back to you. If expect the client to invoke your method with some object A that you sent to them earlier, and instead they send you object B (that you also sent to them earlier), and you don't check it somehow, then you've just opened up a security hole (we'll see an example of this shortly). It may be better to keep such objects in a dictionary on the server side, and have the client send you an index string instead. Doing it that way makes it obvious that they can send you anything they want, and improves the chances that you'll remember to implement the right checks. (This is exactly what PB is doing underneath, with a per-connection dictionary of `Referenceable` objects, indexed by a number).

And, of course, you have to make sure you don't accidentally hand out a reference to the wrong object.

But again, note the vulnerability. If Alice holds a `RemoteReference` to *any* object on the server side that has a `.name` attribute, she can use that name as a spoofed “from” parameter. As a simple example, what if her client code looked like:

```

class ClientThing(pb.Referenceable):
    def join(self):
        d = self.remoteUser.callRemote("joinGroup", "#twisted")
        d.addCallback(self.gotGroup)
    def gotGroup(self, group):
        group.callRemote("send", from_user=group, "hi everybody")

```

This would let her send a message that appeared to come from “#twisted” rather than “Alice”. If she joined a group that happened to be named “bob” (perhaps it is the “How To Be Bob” channel, populated by Alice and countless others, a place where they can share stories about their best impersonating-Bob moments), then she would be able to emit a message that looked like “<bob> says: hi there”, and she has accomplished her lifelong goal.

Argument Typechecking

There are two techniques to close this hole. The first is to have your remotely-invokable methods do type-checking on their arguments: if `Group.remote_send` asserted `isinstance(from_user, User)` then Alice couldn't use non-`User` objects to do her spoofing, and hopefully the rest of the system is designed well enough to prevent her from obtaining access to somebody else's `User` object.

Objects as Capabilities

The second technique is to avoid having the client send you the objects altogether. If they don't send you anything, there is nothing to verify. In this case, you would have to have a per-user-per-group object, in which the `remote_send` method would only take a single message argument. The `UserGroup` object is created with references to the only `User` and `Group` objects that it will ever use, so no lookups are needed:

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

class Group:
    def __init__(self, groupname, allowMattress):
        self.name = groupname
        self.allowMattress = allowMattress
        self.users = []
    def send(self, from_user, message):
        if not self.allowMattress and message.find("mattress") != -1:
            raise ValueError, "Don't say that word"
        for user in self.users:
            user.send("<%s> says: %s" % (from_user.name, message))
    def addUser(self, user):
        self.users.append(user)
```

The only message-sending method Alice has left is `UserGroup.remote_send`, and it only accepts a message: there are no remaining ways to influence the “from” name.

In this model, each remotely-accessible object represents a very small set of capabilities. Security is achieved by only granting a minimal set of abilities to each remote user.

PB provides a shortcut which makes this technique easier to use. The `Viewable` class will be discussed *below* (page 210).

7.5.3 Avatars and Perspectives

In Twisted's *cred* (page 150) system, an “Avatar” is an object that lives on the “server” side (defined here as the side farthest from the human who is trying to get something done) which lets the remote user get something done. The avatar isn't really a particular class, it's more like a description of a role that some object plays, as in “the Foo object here is acting as the user's avatar for this particular service”. Generally, the remote user has some way of getting their avatar to run some code. The avatar object may enforce some security checks, and provide additional data, then call other methods which get things done.

The two pieces in the cred puzzle (for any protocol, not just PB) are: “what serves as the Avatar?”, and “how does the user get access to it?”.

For PB, the first question is easy. The Avatar is a remotely-accessible object which can run code: this is a perfect description of `pb.Referenceable` and its subclasses. We shall defer the second question until the next section.

In the example above, you can think of the `ChatServer` and `Group` objects as a service. The `User` object is the user's server-side representative: everything the user is capable of doing is done by running one of its methods. Anything that the server wants to do to the user (change their group membership, change their name, delete their pet cat, whatever) is done by manipulating the `User` object.

There are multiple `User` objects living in peace and harmony around the `ChatServer`. Each has a different point of view on the services provided by the `ChatServer` and the `Groups`: each may belong to different groups, some might have more permissions than others (like the ability to create groups). These different points of view are called “Perspectives”. This is the origin of the term “Perspective” in “Perspective Broker”: PB provides and controls (i.e. “brokers”) access to Perspectives.

Once upon a time, these local-representative objects were actually called `pb.Perspective`. But this has changed with the advent of the rewritten cred system, and now the more generic term for a local representative object

is an Avatar. But you will still see reference to “Perspective” in the code, the docs, and the module names¹⁵. Just remember that perspectives and avatars are basically the same thing.

Despite all we’ve been *telling you* (page 150) about how Avatars are more of a concept than an actual class, the base class from which you can create your server-side avatar-ish objects is, in fact, named `pb.Avatar`¹⁶. These objects behave very much like `pb.Referenceable`. The only difference is that instead of offering “remote_FOO” methods, they offer “perspective_FOO” methods.

The other way in which `pb.Avatar` differs from `pb.Referenceable` is that the avatar objects are designed to be the first thing retrieved by a cred-using remote client. Just as `PBClientFactory.getRootObject` gives the client access to a `pb.Root` object (which can then provide access to all kinds of other objects), `PBClientFactory.login` gives client access to a `pb.Avatar` object (which can return other references).

So, the first half of using cred in your PB application is to create an Avatar object which implements `perspective_` methods and is careful to do useful things for the remote user while remaining vigilant against being tricked with unexpected argument values. It must also be careful to never give access to objects that the user should not have access to, whether by returning them directly, returning objects which contain them, or returning objects which can be asked (remotely) to provide them.

The second half is how the user gets a `pb.RemoteReference` to your Avatar. As explained *elsewhere* (page 150), Avatars are obtained from a Realm. The Realm doesn’t deal with authentication at all (usernames, passwords, public keys, challenge-response systems, retinal scanners, real-time DNA sequencers, etc). It simply takes an “avatarID” (which is effectively a username) and returns an Avatar object. The Portal and its Checkers deal with authenticating the user: by the time they are done, the remote user has proved their right to access the avatarID that is given to the Realm, so the Realm can return a remotely-controllable object that has whatever powers you wish to grant to this particular user.

For PB, the realm is expected to return a `pb.Avatar` (or anything which implements `pb.IPerspective`, really, but there’s no reason to not return a `pb.Avatar` subclass). This object will be given to the client just like a `pb.Root` would be without cred, and the user can get access to other objects through it (if you let them).

The basic idea is that there is a separate `IPerspective`-implementing object (i.e. the Avatar subclass) (i.e. the “perspective”) for each user, and *only* the authorized user gets a remote reference to that object. You can store whatever permissions or capabilities the user possesses in that object, and then use them when the user invokes a remote method. You give the user access to the perspective object instead of the objects that do the real work.

7.5.4 Perspective Examples

Here is a brief example of using a `pb.Avatar`. Most of the support code is magic for now: we’ll explain it later.

One Client

```
#!/usr/bin/python

from zope.interface import implements

from twisted.spread import pb
from twisted.cred import checkers, portal
from twisted.internet import reactor

class MyPerspective(pb.Avatar):
    def __init__(self, name):
        self.name = name
    def perspective_foo(self, arg):
        print "I am", self.name, "perspective_foo(",arg,") called on", self

class MyRealm:
```

¹⁵We could just go ahead and rename Perspective Broker to be Avatar Broker, but 1) that would cause massive compatibility problems, and 2) “AB” doesn’t fit into the whole sandwich-themed naming scheme nearly as well as “PB” does. If we changed it to AB, we’d probably have to change Banana to be CD (CoderDecoder), and Jelly to be EF (EncapsulatorFragmentor). `twisted.spread` would then have to be renamed `twisted.alphabetsoup`, and then the whole food-pun thing would start all over again.

¹⁶The avatar-ish class is named `pb.Avatar` because `pb.Perspective` was already taken, by the (now obsolete) oldcred perspective-ish class. It is a pity, but it simply wasn’t possible both replace `pb.Perspective` in-place *and* maintain a reasonable level of backwards-compatibility.

```

    implements(portal.IRealm)
    def requestAvatar(self, avatarId, mind, *interfaces):
        if pb.IPerspective not in interfaces:
            raise NotImplementedError
        return pb.IPerspective, MyPerspective(avatarId), lambda:None

p = portal.Portal(MyRealm())
p.registerChecker(
    checkers.InMemoryUsernamePasswordDatabaseDontUse(user1="pass1"))
reactor.listenTCP(8800, pb.PBServerFactory(p))
reactor.run()

```

Source listing — *pb5server.py*

```

#!/usr/bin/python

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

def main():
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 8800, factory)
    defl = factory.login(credentials.UsernamePassword("user1", "pass1"))
    defl.addCallback.connected)
    reactor.run()

def connected(perspective):
    print "got perspective ref:", perspective
    print "asking it to foo(12)"
    perspective.callRemote("foo", 12)

main()

```

Source listing — *pb5client.py*

Ok, so that wasn't really very exciting. It doesn't accomplish much more than the first PB example, and used a lot more code to do it. Let's try it again with two users this time.

Note:

When the client runs `login` to request the Perspective, they can provide it with an optional `client` argument (which must be a `pb.Referenceable` object). If they do, then a reference to that object will be handed to the realm's `requestAvatar` in the `mind` argument.

The server-side Perspective can use it to invoke remote methods on something in the client, so that the client doesn't always have to drive the interaction. In a chat server, the client object would be the one to which "display text" messages were sent. In a board game server, this would provide a way to tell the clients that someone has made a move, so they can update their game boards.

Two Clients

```

#!/usr/bin/python

from zope.interface import implements

from twisted.spread import pb
from twisted.cred import checkers, portal
from twisted.internet import reactor

```

```

class MyPerspective(pb.Avatar):
    def __init__(self, name):
        self.name = name
    def perspective_foo(self, arg):
        print "I am", self.name, "perspective_foo(",arg,") called on", self

class MyRealm:
    implements(portal.IRealm)
    def requestAvatar(self, avatarId, mind, *interfaces):
        if pb.IPerspective not in interfaces:
            raise NotImplementedError
        return pb.IPerspective, MyPerspective(avatarId), lambda:None

p = portal.Portal(MyRealm())
c = checkers.InMemoryUsernamePasswordDatabaseDontUse(user1="pass1",
                                                    user2="pass2")

p.registerChecker(c)
reactor.listenTCP(8800, pb.PBServerFactory(p))
reactor.run()

```

Source listing — *pb6server.py*

```

#!/usr/bin/python

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

def main():
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 8800, factory)
    defl = factory.login(credentials.UsernamePassword("user1", "pass1"))
    defl.addCallback.connected)
    reactor.run()

def connected(perspective):
    print "got perspective1 ref:", perspective
    print "asking it to foo(13)"
    perspective.callRemote("foo", 13)

main()

```

Source listing — *pb6client1.py*

```

#!/usr/bin/python

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

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

def main():

```



```

factory = pb.PBClientFactory()
reactor.connectTCP("localhost", 8800, factory)
defl = factory.login(credentials.UsernamePassword("user2", "pass2"))
defl.addCallback(connected)
reactor.run()

def connected(perspective):
    print "got perspective2 ref:", perspective
    print "asking it to foo(14)"
    perspective.callRemote("foo", 14)

main()

```

Source listing — *pb6client2.py*

While *pb6server.py* is running, try starting *pb6client1*, then *pb6client2*. Compare the argument passed by the `.callRemote()` in each client. You can see how each client gets connected to a different Perspective.

How that example worked

Let's walk through the previous example and see what was going on.

First, we created a subclass called *MyPerspective* which is our server-side Avatar. It implements a *perspective.foo* method that is exposed to the remote client.

Second, we created a realm (an object which implements *IRrealm*, and therefore implements *requestAvatar*). This realm manufactures *MyPerspective* objects. It makes as many as we want, and names each one with the *avatarID* (a username) that comes out of the checkers. This *MyRealm* object returns two other objects as well, which we will describe later.

Third, we created a portal to hold this realm. The portal's job is to dispatch incoming clients to the credential checkers, and then to request Avatars for any which survive the authentication process.

Fourth, we made a simple checker (an object which implements *IChecker*) to hold valid user/password pairs. The checker gets registered with the portal, so it knows who to ask when new clients connect. We use a checker named *InMemoryUsernamePasswordDatabaseDontUse*, which suggests that 1: all the username/password pairs are kept in memory instead of being saved to a database or something, and 2: you shouldn't use it. The admonition against using it is because there are better schemes: keeping everything in memory will not work when you have thousands or millions of users to keep track of, the passwords will be stored in the *.tap* file when the application shuts down (possibly a security risk), and finally it is a nuisance to add or remove users after the checker is constructed.

Fifth, we create a *pb.PBServerFactory* to listen on a TCP port. This factory knows how to connect the remote client to the Portal, so incoming connections will be handed to the authentication process. Other protocols (non-PB) would do something similar: the factory that creates Protocol objects will give those objects access to the Portal so authentication can take place.

On the client side, a *pb.PBClientFactory* is created (as *before* (page 176)) and attached to a TCP connection. When the connection completes, the factory will be asked to produce a Protocol, and it will create a PB object. Unlike the previous chapter, where we used `.getRootObject`, here we use `factory.login` to initiate the credential authentication process. We provide a *credentials* object, which is the client-side agent for doing our half of the authentication process. This process may involve several messages: challenges, responses, encrypted passwords, secure hashes, etc. We give our *credentials* object everything it will need to respond correctly (in this case, a username and password, but you could write a credential that used public-key encryption or even fancier techniques).

`login` returns a *Deferred* which, when it fires, will return a *pb.RemoteReference* to the remote avatar. We can then do `callRemote` to invoke a *perspective.foo* method on that Avatar.

7.5.5 Using Avatars

Avatar Interfaces

The first element of the 3-tuple returned by *requestAvatar* indicates which Interface this Avatar implements. For PB avatars, it will always be *pb.IPerspective*, because that's the only interface these avatars implement.

This element is present because *requestAvatar* is actually presented with a list of possible Interfaces. The question being posed to the Realm is: "do you have an avatar for (*avatarID*) that can implement one of the following

set of Interfaces?”. Some portals and checkers might give a list of Interfaces and the Realm could pick; the PB code only knows how to do one, so we cannot take advantage of this feature.

Logging Out

The third element of the 3-tuple is a zero-argument callable, which will be invoked by the protocol when the connection has been lost. We can use this to notify the Avatar when the client has lost its connection. This will be described in more detail below.

Making Avatars

In the example above, we create Avatars upon request, during `requestAvatar`. Depending upon the service, these Avatars might already exist before the connection is received, and might outlive the connection. The Avatars might also accept multiple connections.

Another possibility is that the Avatars might exist ahead of time, but in a different form (frozen in a pickle and/or saved in a database). In this case, `requestAvatar` may need to perform a database lookup and then do something with the result before it can provide an avatar. In this case, it would probably return a `Deferred` so it could provide the real Avatar later, once the lookup had completed.

Here are some possible implementations of `MyRealm.requestAvatar`:

```
# pre-existing, static avatars
def requestAvatar(self, avatarID, mind, *interfaces):
    assert pb.IPerspective in interfaces
    avatar = self.avatars[avatarID]
    return pb.IPerspective, avatar, lambda:None

# database lookup and unpickling
def requestAvatar(self, avatarID, mind, *interfaces):
    assert pb.IPerspective in interfaces
    d = self.database.fetchAvatar(avatarID)
    d.addCallback(self.doUnpickle)
    return pb.IPerspective, d, lambda:None
def doUnpickle(self, pickled):
    avatar = pickle.loads(pickled)
    return avatar

# everybody shares the same Avatar
def requestAvatar(self, avatarID, mind, *interfaces):
    assert pb.IPerspective in interfaces
    return pb.IPerspective, self.theOneAvatar, lambda:None

# anonymous users share one Avatar, named users each get their own
def requestAvatar(self, avatarID, mind, *interfaces):
    assert pb.IPerspective in interfaces
    if avatarID == checkers.ANONYMOUS:
        return pb.IPerspective, self.anonAvatar, lambda:None
    else:
        return pb.IPerspective, self.avatars[avatarID], lambda:None

# anonymous users get independent (but temporary) Avatars
# named users get their own persistent one
def requestAvatar(self, avatarID, mind, *interfaces):
    assert pb.IPerspective in interfaces
    if avatarID == checkers.ANONYMOUS:
        return pb.IPerspective, MyAvatar(), lambda:None
    else:
        return pb.IPerspective, self.avatars[avatarID], lambda:None
```

The last example, note that the new `MyAvatar` instance is not saved anywhere: it will vanish when the connection is dropped. By contrast, the avatars that live in the `self.avatars` dictionary will probably get persisted into the `.tap` file along with the Realm, the Portal, and anything else that is referenced by the top-level Application object. This is an easy way to manage saved user profiles.

Connecting and Disconnecting

It may be useful for your Avatars to be told when remote clients gain (and lose) access to them. For example, and Avatar might be updated by something in the server, and if there are clients attached, it should update them (through the “mind” argument which lets the Avatar do `callRemote` on the client).

One common idiom which accomplishes this is to have the Realm tell the avatar that a remote client has just attached. The Realm can also ask the protocol to let it know when the connection goes away, so it can then inform the Avatar that the client has detached. The third member of the `requestAvatar` return tuple is a callable which will be invoked when the connection is lost.

```
class MyPerspective(pb.Avatar):
    def __init__(self):
        self.clients = []
    def attached(self, mind):
        self.clients.append(mind)
        print "attached to", mind
    def detached(self, mind):
        self.clients.remove(mind)
        print "detached from", mind
    def update(self, message):
        for c in self.clients:
            c.callRemote("update", message)

class MyRealm:
    def requestAvatar(self, avatarID, mind, *interfaces):
        assert pb.IPerspective in interfaces
        avatar = self.avatars[avatarID]
        avatar.attached(mind)
        return pb.IPerspective, avatar, lambda a=avatar:a.detached(mind)
```

Viewable

Once you have `IPerspective` objects (i.e. the Avatar) to represent users, the `Viewable` class can come into play. This class behaves a lot like `Referenceable`: it turns into a `RemoteReference` when sent over the wire, and certain methods can be invoked by the holder of that reference. However, the methods that can be called have names that start with `view_` instead of `remote_`, and those methods are always called with an extra `perspective` argument that points to the Avatar through which the reference was sent:

```
class Foo(pb.Viewable):
    def view_doFoo(self, perspective, arg1, arg2):
        pass
```

This is useful if you want to let multiple clients share a reference to the same object. The `view_` methods can use the “perspective” argument to figure out which client is calling them. This gives them a way to do additional permission checks, do per-user accounting, etc.

This is the shortcut which makes per-user-per-group capability objects much easier to use. Instead of creating such per-(user,group) objects, you just have per-group objects which inherit from `pb.Viewable`, and give the user references to them. The local `pb.Avatar` object will automatically show up as the “perspective” argument in the `view_*` method calls, give you a chance to involve the Avatar in the process.

Chat Server with Avatars

Combining all the above techniques, here is an example chat server which uses a fixed set of identities (say, for the three members of your bridge club, who hang out in “#NeedAFourth” hoping that someone will discover your server, guess somebody’s password, break in, join the group, and also be available for a game next saturday afternoon).

```

#!/usr/bin/python

from zope.interface import implements

from twisted.cred import portal, checkers
from twisted.spread import pb
from twisted.internet import reactor

class ChatServer:
    def __init__(self):
        self.groups = {} # indexed by name

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

class ChatRealm:
    implements(portal.IRealm)
    def requestAvatar(self, avatarID, mind, *interfaces):
        assert pb.IPerspective in interfaces
        avatar = User(avatarID)
        avatar.server = self.server
        avatar.attached(mind)
        return pb.IPerspective, avatar, lambda a=avatar:a.detached(mind)

class User(pb.Avatar):
    def __init__(self, name):
        self.name = name
    def attached(self, mind):
        self.remote = mind
    def detached(self, mind):
        self.remote = None
    def perspective_joinGroup(self, groupname, allowMattress=True):
        return self.server.joinGroup(groupname, self, allowMattress)
    def send(self, message):
        self.remote.callRemote("print", message)

class Group(pb.Viewable):
    def __init__(self, groupname, allowMattress):
        self.name = groupname
        self.allowMattress = allowMattress
        self.users = []
    def addUser(self, user):
        self.users.append(user)
    def view_send(self, from_user, message):
        if not self.allowMattress and message.find("mattress") != -1:
            raise ValueError, "Don't say that word"
        for user in self.users:
            user.send("<%s> says: %s" % (from_user.name, message))

realm = ChatRealm()
realm.server = ChatServer()
checker = checkers.InMemoryUsernamePasswordDatabaseDontUse()
checker.addUser("alice", "1234")
checker.addUser("bob", "secret")

```

```

checker.addUser("carol", "fido")
p = portal.Portal(realm, [checker])

reactor.listenTCP(8800, pb.PBServerFactory(p))
reactor.run()

```

Source listing — *chatserver.py*

Notice that the client uses `perspective_joinGroup` to both join a group and retrieve a `RemoteReference` to the `Group` object. However, the reference they get is actually to a special intermediate object called a `pb.ViewPoint`. When they do `group.callRemote("send", "message")`, their avatar is inserted into the argument list that `Group.view_send` actually sees. This lets the group get their username out of the `Avatar` without giving the client an opportunity to spoof someone else.

The client side code that joins a group and sends a message would look like this:

```

#!/usr/bin/python

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

class Client(pb.Referenceable):

    def remote_print(self, message):
        print message

    def connect(self):
        factory = pb.PBClientFactory()
        reactor.connectTCP("localhost", 8800, factory)
        defl = factory.login(credentials.UsernamePassword("alice", "1234"),
                             client=self)
        defl.addCallback(self.connected)
        reactor.run()

    def connected(self, perspective):
        print "connected, joining group #lookingForFourth"
        # this perspective is a reference to our User object
        d = perspective.callRemote("joinGroup", "#lookingForFourth")
        d.addCallback(self.gotGroup)

    def gotGroup(self, group):
        print "joined group, now sending a message to all members"
        # 'group' is a reference to the Group object (through a ViewPoint)
        d = group.callRemote("send", "You can call me Al.")
        d.addCallback(self.shutdown)

    def shutdown(self, result):
        reactor.stop()

Client().connect()

```

Source listing — *chatclient.py*

Chapter 8

Manual Pages

8.1 MANHOLE.1

8.1.1 NAME

manhole - Connect to a Twisted Manhole service

8.1.2 SYNOPSIS

manhole

8.1.3 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.

8.1.4 AUTHOR

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

8.1.5 REPORTING BUGS

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

8.1.6 COPYRIGHT

Copyright ©2000 Matthew W. Lefkowitz This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

8.2 MKTAP.1

8.2.1 NAME

mktap - create twisted.servers

8.2.2 SYNOPSIS

```
mktap [options] apptype [application_option]...
mktapapptype -help
```

8.2.3 DESCRIPTION

The *-help* prints out a usage message to standard output. *-debug*, *-d* Show debug information for plugin loading. *-progress*, *-p* Show progress information for plugin loading. *-encrypted*, *-e* Encrypt file before writing (will make the extension of the resultant file begin with 'e'). *-uid*, *-u*<uid> Application belongs to this uid, and should run with its permissions. *-gid*, *-d*<gid> Application belongs to this gid, and should run with its permissions. *-append*, *-a*<file> Append given servers to given file, instead of creating a new one. File should be be a tap file. *-appname*, *-n*<name> Use the specified name as the process name when the application is run with *twistd(1)*. This option also causes some initialization code to be duplicated when *twistd(1)* is run. *-type*, *-t*<type> Specify the output file type. Available types are: pickle - (default) Output as a python pickle file. xml - Output as a .tax XML file. source - Output as a .tas (AOT Python source) file. *apptype* Can be 'web', 'portforward', 'toc', 'coil', 'words', 'manhole', 'im', 'news', 'socks', 'telnet', 'parent', 'sibling', 'ftp', and 'mail'. Each of those support different options.

8.2.4 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>

8.2.5 web options

-u, *-user* Makes a server with ~/public.html and ~/.twistd-web-pb support for users. *-personal* Instead of generating a webserver, generate a ResourcePublisher which listens on ~/.twistd-web-pb *-path*<path><path> is either a specific file or a directory to be set as the root of the web server. Use this if you have a directory full of HTML, cgi, php3, epy, or rpy files or any other files that you want to be served up raw. *-p*, *-port*<port><port> is a number representing which port you want to start the server on. *-m*, *-mime_type*<mimetype><mimetype> is the default MIME type to use for files in a *-path* web server when none can be determined for a particular extension. The default is 'text/html'. *-allow_ignore_ext* Specify whether or not a request for 'foo' should return 'foo.ext'. Default is off. *-ignore_ext.<extension>* Specify that a request for 'foo' should return 'foo.<extension>'. *-t*, *-telnet*<port> Run a telnet server on <port>, for additional configuration later. *-i*, *-index*<name> Use an index name other than "index.html" *-https*<port> Port to listen on for Secure HTTP. *-c*, *-certificate*<filename> SSL certificate to use for HTTPS. [default: server.pem] *-k*, *-privkey*<filename> SSL certificate to use for HTTPS. [default: server.pem] *-processor*<ext>=<class name> Adds a processor to those file names. (Only usable if after *-path*) *-resource-script*<script name> Sets the root as a resource script. This script will be re-evaluated on every request. This creates a web.tap file that can be used by twistd. If you specify no arguments, it will be a demo webserver that has the Test class from twisted.web.test in it.

8.2.6 toc options

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

8.2.7 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)*

8.2.8 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>*

8.2.9 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>*

8.2.10 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

8.2.11 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>*

8.2.12 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>*

8.2.13 AUTHOR

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

8.2.14 REPORTING BUGS

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

8.2.15 COPYRIGHT

Copyright ©2000 Matthew W. Lefkowitz This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

8.2.16 SEE ALSO

twistd(1)

8.3 TAP2DEB.1

8.3.1 NAME

tap2deb - create Debian packages which wrap .tap files

8.3.2 SYNOPSIS

tap2deb [options]

8.3.3 DESCRIPTION

Create a ready to upload Debian package in “.build” -*u*, -*unsigned* do not sign the Debian package -*t*, -*tapfile*<tapfile> Build the application around the given .tap (default twistd.tap) -*y*, -*type*<type> The configuration has the given type . Allowable types are *tap*, *source*, *xml* and *python*. The first three types are *mkcap(1)* output formats, while the last one is a manual building of application (see *twistd(1)*, the -*y* option). -*p*, -*protocol*<protocol> The name of the protocol this will be used to serve. This is intended as a part of the description. Default is the name of the tapfile, minus any extensions. -*d*, -*debfile*<debfile> The name of the debian package. Default is 'twisted-' + protocol. -*V*, -*set-version*<version> The version of the Debian package. The default is 1.0 -*e*, -*description*<description> The one-line description. Default is uninteresting. -*l*, -*long_description*<long_description> A multi-line description. Default is explanation about this being an automatic package created from tap2deb. -*m*, -*maintainer*<maintainer> The maintainer, as “Name Lastname <email address>”. This will go in the meta-files, as well as be used as the id to sign the package. -*v*, -*version* Output version information and exit.

8.3.4 AUTHOR

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

8.3.5 REPORTING BUGS

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

8.3.6 COPYRIGHT

Copyright ©2000 Matthew W. Lefkowitz This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

8.3.7 SEE ALSO

mkcap(1)

8.4 TAP2RPM.1

8.4.1 NAME

tap2rpm - create RPM packages which wrap .tap files

8.4.2 SYNOPSIS

tap2rpm [options]

8.4.3 DESCRIPTION

Create a set of RPM/SRPM packages in the current directory *-u*, *-unsigned* do not sign the RPM package *-t*, *-tapfile*<tapfile> Build the application around the given .tap (default twisted.tap) *-y*, *-type*<type> The configuration has the given type . Allowable types are *tap*, *source*, *xml* and *python*. The first three types are *mktrap(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*, *-rpmfile*<rpmfile> The name of the RPM package. Default is 'twisted-' + protocol. *-V*, *-set-version*<version> The version of the RPM package. The default is 1.0 *-e*, *-description*<description> The one-line description. Default is uninteresting. *-l*, *-long-description*<long-description> A multi-line description. Default is explanation about this being an automatic package created from tap2rpm. *-m*, *-maintainer*<maintainer> The maintainer, as "Name Lastname <email address>". This will go in the meta-files, as well as be used as the id to sign the package. *-v*, *-version* Output version information and exit.

8.4.4 AUTHOR

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

8.4.5 REPORTING BUGS

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

8.4.6 COPYRIGHT

Copyright ©2000 Matthew W. Lefkowitz This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

8.4.7 SEE ALSO

mktrap(1)

8.5 TAPCONVERT.1

8.5.1 NAME

tapconvert - convert Twisted configurations from one format to another

8.5.2 SYNOPSIS

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

8.5.3 DESCRIPTION

The *-help* prints out a usage message to standard output. *-in, -i<input file>* The name of the input configuration. *-out, -o<output file>* The name of the output configuration. *-typein, -f<input type>* The type of the input file. Can be either 'guess', 'python', 'pickle', 'xml', or 'source'. Default is 'guess'. *-typeout, -t<output type>* The type of the output file. Can be either 'pickle', 'xml', or 'source'. Default is 'source'. *-decrypt, -d* Decrypt input. *-encrypt, -e* Encrypt output.

8.5.4 AUTHOR

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

8.5.5 REPORTING BUGS

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

8.5.6 COPYRIGHT

Copyright ©2000 Matthew W. Lefkowitz This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

8.5.7 SEE ALSO

mktap(1)

8.6 TRIAL.1

8.6.1 NAME

trial - run unit tests

8.6.2 SYNOPSIS

```
trial [options] [[file—package—module—TestCase—testmethod]...]
trial -help
```

8.6.3 DESCRIPTION

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

trial will take either filenames or fully qualified Python names as arguments. Thus 'trial myproject/foo.py', 'trial myproject.foo' and 'trial myproject.foo.SomeTestCase.test_method' are all valid ways to invoke trial. *-b*, *-debug* Run the tests in the Python debugger. Also does post-mortem debugging on exceptions. *-B*, *-debug-stacktraces* Report Deferred creation and callback stack traces *-coverage* Generate coverage information in `_trial_temp/coverage/`. Requires Python 2.3 or higher. *-disablegc* Disable the garbage collector. I don't know why this is in trial. *-e*, *-rerrors* Print tracebacks to standard output as soon as they occur *-force-gc* Run `gc.collect()` before and after each test case. This can be used to isolate errors that occur when objects get collected. This option would be the default, except it makes tests run about ten times slower. *-h*, *-help* Print a usage message to standard output, then exit. *-help-reporters* Print a list of valid reporters to standard output, then exit. *-help-reactors* List the names of possibly available reactors. *-l*, *-logfile*<logfile> Direct the log to a different file. The default file is 'test.log'. <logfile> is relative to `_trial_temp`. *-n*, *-dry-run* Go through all the tests and make them pass without running. *-N*, *-no-recurse* By default, trial recurses through packages to find every module inside every subpackage. Unless, that is, you specify this option. *-nopm* Don't automatically jump into debugger for post-mortem analysis of exceptions. Only usable in conjunction with *-debug*. *-profile* I don't know what this option does. *-r*, *-reactor*<reactor> Choose which reactor to use. See *-help-reactors* for a list. *-recursionlimit* Set Python's recursion limit. I don't know why this is in trial. *-reporter* Select the reporter to use for Trial's output. Use the *-help-reporters* option to see a list of valid reporters. *-spew* Print an insanely verbose log of everything that happens. Useful when debugging freezes or locks in complex code. *-tbformat*<format> Format to display tracebacks with. Acceptable values are 'default', 'brief' and 'verbose'. 'brief' produces tracebacks that play nicely with Emacs' GUD. *-temp-directory* <directory> WARNING: Do not use this options unless you know what you are doing. By default, trial creates a directory called `_trial_temp` under the current working directory. When trial runs, it first *deletes* this directory, then creates it, then changes into the directory to run the tests. The log file and any coverage files are stored here. Use this option if you wish to have trial run in a directory other than `_trial_temp`. Be warned, trial will *delete* the directory before re-creating it. *-testmodule* <filename> Ask trial to look into <filename> and run any tests specified using the Emacs-style buffer variable 'test-case-name'. *-u*, *-until-failure* Keep looping the tests until one of them raises an error or a failure. This is particularly useful for reproducing intermittent failures. *-z*, *-random* [<seed>] Run the tests in random order using the specified seed.

8.6.4 AUTHOR

Written by Jonathan M. Lange

8.6.5 REPORTING BUGS

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

8.6.6 COPYRIGHT

Copyright ©2003-2006 Twisted Matrix Laboratories This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

8.7 TWISTD.1

8.7.1 NAME

twistd - run Twisted applications (TACs, TAPs)

8.7.2 SYNOPSIS

twistd [options]

8.7.3 DESCRIPTION

Read an twisted.application.service.Application out of a file and runs it. *-n*, *-nodaemon* Don't daemonize (stay in foreground) *-q*, *-quiet* be a little more quiet *-p*, *-profile* Run profiler *-b*, *-debug* Run the application in the Python Debugger (implies nodaemon option). Sending a SIGUSR2 signal to the process will drop it into the debugger. *-o*, *-no_save* Do not save shutdown state *-originalname* Behave as though the specified Application has no process name set, and run with the standard process name (the python binary in most cases). *-l*, *-logfile<logfile>* Log to a specified file, - for stdout (default twistd.log). The log file will be rotated on SIGUSR1. *-pidfile<pidfile>* Save pid in specified file (default twistd.pid) *-chroot<directory>* Chroot to a supplied directory before running (default - don't chroot). Chrooting is done before changing the current directory. *-d*, *-rundir<directory>* Change to a supplied directory before running (default .) *-r*, *-reactor<reactor>* Choose which reactor to use. See *-help-reactors* for a list. *-help-reactors* List the names of possibly available reactors. *-spew* Write an extremely verbose log of everything that happens. Useful for debugging freezes or locks in complex code. *-f*, *-file<tap file>* Read the given .tap file (default twistd.tap) *-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*. This option implies *-no_save*. *-g*, *-plugin<plugin name>* Read config.tac from a plugin package, as with *-y*. *-syslog* Log to syslog, not to file. *-prefix<prefix>* Use the specified prefix when logging to logfile. Default is "twisted". Note that if *twistd* is run as root, the working directory is *not* searched for Python modules.

8.7.4 AUTHOR

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

8.7.5 REPORTING BUGS

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

8.7.6 COPYRIGHT

Copyright ©2003 Matthew W. Lefkowitz This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

8.7.7 SEE ALSO

mktap(1)

Chapter 9

Appendix

9.1 The Twisted FAQ

9.1.1 General

What is “Twisted”?

Please see Twisted¹

Why should I use Twisted?

See The Twisted Advantage²

I have a problem “getting” Twisted.

Did you check the HOWTO collection? There are so many documents there that they might overwhelm you... try starting from the index, reading through the overviews and seeing if there seems to be a chapter which explains what you need to. You can try reading the PostScript or PDF formatted books, inside the distribution. And, remember, the source will be with you... always.

Why are there so many parts and subprojects? Isn't Twisted just Twisted?

As of version 2.0, Twisted was split up into many subprojects, because it was getting too much to handle in a monolithic release, and we believe breaking the project into smaller chunks will help people understand the things they need to understand (There used to be a FAQ entry here asking “Why is Twisted so big?”). More information is available in the Split FAQ³.

9.1.2 Stability

Does the 1.0 release mean that all of Twisted's APIs are stable?

No, only specific parts of Twisted are stable, i.e. we only promise backwards compatibility for some parts of Twisted. While these APIs may be extended, they will not change in ways that break existing code that uses them.

While other parts of Twisted are not stable, we will however do our best to make sure that there is backwards compatibility for these parts as well. In general, the more the module or package are used, and the closer they are to being feature complete, the more we will concentrate on providing backwards compatibility when API changes take place.

¹<http://twistedmatrix.com/products/twisted>

²<http://twistedmatrix.com/services/twisted-advantage>

³<http://twistedmatrix.com/products/splitfaq>

Which parts of Twisted are stable?

Only modules explicitly marked as such can be considered stable. Semi-stable modules may change, but not in a large way and some sort of backwards-compatibility will probably be provided. If no comment about API stability is present, assume the module is unstable.

In Twisted 1.1, *most of twisted.internet, .cred and .application are completely stable* (excepting of course code marked as deprecated).

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

9.1.3 Installation**I run mktap (from site-packages/twisted/scripts/mktap.py) and nothing happens!**

Don't run scripts out of `site-packages`. The Windows installer should install executable scripts to someplace like `C:\Python22\scripts\`, *nix installers put them in `$PREFIX/bin`, which should be in your `$PATH`.

Why do the Debian packages for Alphas and Release Candidates have weird versions containing old version numbers?

An example: `1.0.6+1.0.7rc1-1`

In Debian versioning, `1.0.7rc1` is *greater than* `1.0.7`. This means that if you install a package with `Version: 1.0.7rc1`, and then that package gets a new version `1.0.7`, `apt` will not upgrade it for you, because `1.0.7` looks like an older version. So, we prefix the previous version to the actual version. `1.0.6+1.0.7rc1` is *less than* `1.0.7`.

9.1.4 Core Twisted**How can I access self.factory from my Protocol's __init__?**

You can't. A Protocol doesn't have a Factory when it is created. Instead, you should probably be doing that in your Protocol's `connectionMade` method.

Similarly you shouldn't be doing "real" work, like connecting to databases, in a Factory's `__init__` either. Instead, do that in `startFactory`.

See *Writing Servers* (page 13) and *Writing Clients* (page 17) for more details.

Where can I find out how to write Twisted servers?

Try *Writing Servers* (page 13).

When I try to install my reactor, I get errors about a reactor already being installed. What gives?

Here's the rule - installing a reactor should always be the *first* thing you do, and I do mean first. Importing other stuff before you install the reactor can break your code.

Tkinter and wxPython support, as they do not install a new reactor, can be done at any point, IIRC.

twistd won't load my .tap file! What's this Ephemeral nonsense?

When the pickled application state cannot be loaded for some reason, it is common to get a rather opaque error like so:

```
% twistd -f test2.tap
```

```
Failed to load application: global name 'initRun' is not defined
```

The rest of the error will try to explain how to solve this problem, but a short comment first: this error is indeed terse - but there is probably more data available elsewhere - namely, the `twistd.log` file. Open it up to see the full exception.

The error might also look like this:

```
Failed to load application: <twisted.persisted.styles.Ephemeral instance at 0x82450a4> is not safe for unpickling
```

To load a `.tap` file, as with any unpickling operation, all the classes used by all the objects inside it must be accessible at the time of the reload. This may require the `PYTHONPATH` variable to have the same directories as were available when the application was first pickled.

A common problem occurs in single-file programs which define a few classes, then create instances of those classes for use in a server of some sort. If the class is used directly, the name of the class will be recorded in the `.tap` file as something like `__main__.MyProtocol`. When the application is reloaded, it will look for the class definition in `__main__`, which probably won't have it. The unpickling routines need to know the module name, and therefore the source file, from which the class definition can be loaded.

The way to fix this is to import the class from the same source file that defines it: if your source file is called `myprogram.py` and defines a class called `MyProtocol`, you will need to do a `from myprogram import MyProtocol` before (and in the same namespace as) the code that references the `MyProtocol` class. This makes it important to write the module cleanly: doing an `import myprogram` should only define classes, and should not cause any other subroutines to get run. All the code that builds the `Application` and saves it out to a `.tap` file must be inside an `if __name__ == '__main__':` clause to make sure it is not run twice (or more).

When you import the class from the module using an “external” name, that name will be recorded in the pickled `.tap` file. When the `.tap` is reloaded by `twisted`, it will look for `myprogram.py` to provide the definition of `MyProtocol`.

Here is a short example of this technique:

```
# file dummy.py
from twisted.internet import protocol
class Dummy(protocol.Protocol): pass
if __name__ == '__main__':
    from twisted.application import service, internet
    a = service.Application("dummy")
    import dummy
    f = protocol.Factory()
    f.protocol = dummy.Dummy # Note! Not "Dummy"
    internet.TCPServer(2000, f).setServiceParent(a)
    a.save()
```

I get “Interrupted system call” errors when I use `os.popen2`. How do I read results from a sub-process in Twisted?

You should be using `reactor.spawnProcess` (see `interfaces.IReactorProcess.spawnProcess`). There's also a convenience function, `getProcessOutput`, in `twisted.internet.utils`.

Why don't my `spawnProcess` programs see my environment variables?

`spawnProcess` defaults to clearing the environment of child processes as a security feature. You can either provide a dictionary with exactly the name-value pairs you want the child to use, or you can simply pass in `os.environ` to inherit the complete environment.

My `Deferred` or `DeferredList` never fires, so my program just mysteriously hangs! What's wrong?

It really depends on what your program is doing, but the most common cause is this: it *is* firing – but it's an error, not a success, and you have forgotten to add an *errback* (page 226), so nothing happens. Always add *errbacks*!

The reason `Deferred` can't automatically show your errors is because a `Deferred` can still have callbacks and *errbacks* added to it even after a result is available – so we have no reasonable place to put a logging call that wouldn't result in spurious *tracebacks* that *are* handled later on. There is a facility for printing *tracebacks* when the `Deferreds` are garbage collected – call `defer.setDebugging(True)` to enable it.

My exceptions and *tracebacks* aren't getting printed!

See previous question.

How do I use Deferreds to make my blocking code non-blocking?

You don't. Deferreds don't magically turn a blocking function call into a non-blocking one. A Deferred is just a simple object that represents a *deferred result*, with methods to allow convenient adding of callbacks. (This is a common misunderstanding; suggestions on how to make this clearer in the *Deferred Execution* (page 99) howto are welcome!)

If you have blocking code that you want to use non-blockingly in Twisted, either rewrite it to be non-blocking, or run it in a thread. There is a convenience function, `deferToThread`, to help you with the threaded approach – but be sure to read *Using Threads in Twisted* (page 133).

I get “exceptions.ValueError: signal only works in main thread” when I try to run my Twisted program! What’s wrong?

The default reactor, by default, will install signal handlers to catch events like Ctrl-C, SIGTERM, and so on. However, you can't install signal handlers from non-main threads in Python, which means that `reactor.run()` will cause an error. Pass the `installSignalHandlers=0` keyword argument to `reactor.run` to work around this.

I’m trying to stop my program with `sys.exit()`, but Twisted seems to catch it! How do I exit my program?

Use `reactor.stop()` instead. This will cleanly shutdown the reactor.

How do I find out the IP address of the other end of my connection?

The `.transport` object (which implements the `ITransport` interface) offers a pair of methods named `getPeer` and `getHost`. `getPeer` will give you a tuple that describes the address of the system at the other end of the connection. For example:

```
class MyProtocol(protocol.Protocol):
    def connectionMade(self):
        print "connection from", self.transport.getPeer()
```

Why don’t Twisted’s network methods support Unicode objects as well as strings?

In general, such methods (eg `FileDescriptor`'s `write`) are designed to send bytes over the network. These methods use non-Unicode string objects as a container for the bytes that they send and receive.

Unicode objects are not byte-based and are an abstraction used for representing strings of human readable text. In order to send Unicode strings using these methods, you should explicitly specify a byte-based encoding for them, for example: `s.encode("utf-8")` and explicitly decode them at the receiving end.

Twisted cannot choose an encoding for you at this level: your encoding choice will be protocol specific and may need to be specified in the message you send (for example, HTTP headers include a encoding specification).

For a more complete discussion of the distinction between Unicode strings and specific encodings of Unicode strings, see the following articles:

- Dan Sugalski's What the heck is: A string⁴; and
- Joel Spolsky's The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)⁵.

9.1.5 Requests and Contributing**Twisted is cool, but I need to add more functionality.**

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

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

Use unified diff. Either use `svn diff` or, better yet, make a clean checkout and use `diff -uRN` between them. Make sure your patch applies cleanly. In your post to the mailing list, make sure it is inlined and without any word wrapping.

⁴<http://www.sidhe.org/~dan/blog/archives/000255.html>

⁵<http://www.joelonsoftware.com/articles/Unicode.html>

And to whom do I send it?

Add it to the bug tracker⁶, and if it's an urgent or important issue you may want to tell the mailing list⁷. about the issue you added

My company would love to use Twisted, but it's missing feature X, can you implement it?

You have 3 options:

- Pay one of the Twisted developers to implement the feature.
- Implement the feature yourself.
- Add a feature request to our bug tracker. We will try to implement the feature, but there are no guarantees when and if this will happen.

9.1.6 Documentation**Twisted really needs documentation for X, Y or Z - how come it's not documented?.**

Twisted's documentation is a work in progress, and one that we would appreciate assistance with. If you notice a gap or flaw in the documentation, please file a bug in the Twisted bug tracker⁸ and mark it as having topic 'documentation'. Patches appreciated.

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

Now you can, with Lore⁹.

9.1.7 Communicating with us**There's a bug in Twisted. Where do I report it?**

Unless it is a show-stopper bug, we usually won't fix it if it's already fixed in Subversion¹⁰, so check if it is fixed there. If it is not fixed in Subversion, you should add it to the bug tracker¹¹, including pertinent information about the bug (hopefully as much information needed to reproduce it: OS, Subversion versions of any important files, Python version, code you wrote or things you did to trigger the bug, etc. If the bug appears to be severe, you should also raise it on the mailing list¹², with a pointer to the issue already filed in the bug tracker.

Where do I go for help?

Ask for help where the Twisted team hangs out¹³

How do I e-mail a Twisted developer?

First, note that in many cases this is the wrong thing to do: if you have a question about a part of Twisted, it's usually better to e-mail the mailing list. However, the preferred e-mail addresses for all Twisted developers are listed in the file "CREDITS" in the Subversion repository.

⁶<http://twistedmatrix.com/bugs/>

⁷<http://twistedmatrix.com/cgi-bin/mailman/listinfo/twisted-python>

⁸<http://twistedmatrix.com/bugs/>

⁹<http://twistedmatrix.com/projects/lore>

¹⁰<http://twistedmatrix.com/developers/cvs>

¹¹<http://twistedmatrix.com/bugs/>

¹²<http://twistedmatrix.com/cgi-bin/mailman/listinfo/twisted-python>

¹³<http://twistedmatrix.com/services/online-help>

9.2 Twisted Glossary

adaptee An object that has been adapted, also called “original”. See *Adapter* (page 226).

Adapter An object whose sole purpose is to implement an Interface for another object. See *Interfaces and Adapters* (page 144).

Application A `twisted.application.service.Application`. There are HOWTOs on *creating and manipulating* (page 139) them as a system-administrator, as well as *using* (page 155) them in your code.

Avatar (from *Twisted Cred* (this page)) business logic for specific user. For example, in *PB* (page 227) these are perspectives, in *pop3* these are mailboxes, and so on.

Banana The low-level data marshalling layer of *Twisted Spread* (page 227). See `twisted.spread.banana`.

Broker A `twisted.spread.pb.Broker`, the object request broker for *Twisted Spread* (page 227).

cache A way to store data in readily accessible place for later reuse. Caching data is often done because the data is expensive to produce or access. Caching data risks being stale, or out of sync with the original data.

component A special kind of (persistent) *Adapter* that works with a `twisted.python.components.Componentized`. See also *Interfaces and Adapters* (page 144).

Componentized A *Componentized* object is a collection of information, separated into domain-specific or role-specific instances, that all stick together and refer to each other. Each object is an *Adapter*, which, in the context of *Componentized*, we call “components”. See also *Interfaces and Adapters* (page 144).

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 17).

Consumer An object that consumes data from a *Producer* (page 227). See `twisted.internet.interfaces.IConsumer`.

Cred Twisted’s authentication API, `twisted.cred`. See *Introduction to Twisted Cred* (page 150) and *Twisted Cred usage* (page 200).

credentials A username/password, public key, or some other information used for authentication.

credential checker Where authentication actually happens. See `ICredentialChecker`.

CVSToys A nifty set of tools for CVS, available at <http://twistedmatrix.com/users/acapnotic/wares/code/CVSToys/>.

Deferred A instance of `twisted.internet.defer.Deferred`, an abstraction for handling chains of callbacks and error handlers (“errbacks”). See the *Deferring Execution* (page 99) 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 227)”. See *Introduction to Twisted Enterprise* (page 168) and *Twisted Enterprise Row Objects* (page 170) for more details.

errback A callback attached to a *Deferred* (this page) with `.addErrback` to handle errors.

Factory In general, an object that constructs other objects. In Twisted, a *Factory* usually refers to a `twisted.internet.protocol.Factory`, which constructs *Protocol* (page 227) instances for incoming or outgoing connections. See *Writing Servers* (page 13) and *Writing Clients* (page 17).

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

im, t-im Abbreviation of “(Twisted) *Instance Messenger* (page 227)”.

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 227) with *Twisted Words* (page 228). See `twisted.im`.

Interface A class that defines and documents methods that a class conforming to that interface needs to have. A collection of core `twisted.internet` interfaces can be found in `twisted.internet.interfaces`. See also *Interfaces and Adapters* (page 144).

Jelly The serialization layer for *Twisted Spread* (this page), although it can be used separately from Twisted Spread as well. It is similar in purpose to Python's standard `pickle` module, but is more network-friendly, and depends on a separate marshaller (*Banana* (page 226), in most cases). See `twisted.spread.jelly`.

Lore Lore¹⁴ is Twisted's documentation system. The source format is a subset of XHTML, and output formats include HTML and LaTeX.

Manhole A debugging/administration interface to a Twisted application.

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

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

Nevow The successor to *Woven* (page 228), a web framework available at nevow.com¹⁵.

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

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

Portal Glues *credential checkers* (page 226) and *realm* (this page)s together.

Producer An object that generates data a chunk at a time, usually to be processed by a *Consumer* (page 226). 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 13) and *Writing Clients* (page 17).

PSU There is no PSU.

Reactor The core event-loop of a Twisted application. See *Reactor Basics* (page 89).

Reality See "*Twisted Reality* (page 228)"

realm (in *Twisted Cred* (page 226)) stores *avatars* (page 226) and perhaps general business logic. See `IRealm`.

Resource A `twisted.web.resource.Resource`, which are served by Twisted Web. Resources can be as simple as a static file on disk, or they can have dynamically generated content.

ROW Relational Object Wrapper, an object-oriented interface to a relational database. See *Twisted Enterprise Row Objects* (page 170).

Service A `twisted.application.service.Service`. See *Application howto* (page 155) for a description of how they relate to *Applications* (page 226).

Spread Twisted Spread¹⁶ is Twisted's remote-object suite. It consists of three layers: *Perspective Broker* (this page), *Jelly* (this page) and *Banana*. (page 226) See *Writing Applications with Perspective Broker* (page 173).

SUX Small Uncomplicated XML, Twisted's simple XML parser written in pure Python. See `twisted.protocols.sux`.

TAP Twisted Application Pickle, or simply just a *Twisted Application*. A serialised application that created with `mktap` and runnable by `twistd`. See *Using the Utilities* (page 139).

Trial `twisted.trial`, Twisted's unit-testing framework, modelled after `pyunit`¹⁷. See also *Writing tests for*

¹⁴<http://twistedmatrix.com/projects/lore/>

¹⁵<http://nevow.com/>

¹⁶<http://twistedmatrix.com/products/spread>

¹⁷<http://pyunit.sourceforge.net/>

Twisted code (page 166).

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

Twisted Reality In days of old, the Twisted Reality¹⁸ 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 159).

Words Twisted Words is a multi-protocol chat server that uses the *Perspective Broker* (page 227) protocol as its native communication style. See `twisted.words`.

Woven Web Object Visualization Environment. A deprecated web templating system based on XML and the Model-View-Controller design pattern. This has been deprecated in favor of Nevow¹⁹.

9.3 Banana Protocol Specifications

9.3.1 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.

9.3.2 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 significant bit first. For example length 4674 will be sent as `0x42 0x24`. For certain element types the length will be omitted (e.g. float) or have a different meaning (it is the actual value of integer elements).

Following the length is a delimiter byte, which tells us what kind of element this is. Depending on the element type, there will then follow the number of bytes specified in the length. The byte's high-bit will always be set, so that we can differentiate between it and the length (since the length bytes use 128-base, their high bit will never be set).

9.3.3 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 $0 \leq N \leq 2147483647$.

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 $0 \geq -N \geq -2147483648$.

Float - 0x84 The next 8 bytes are the float encoded in IEEE 754 floating-point “double format” bit layout. No length bytes should have been defined.

Large Integer – 0x85 The value of this element is the positive large integer N. Following bytes are not part of this element. Large integers have no size limitation.

Large Negative Integer – 0x86 The value of this element is the negative large integer $-N$. Following bytes are not part of this element. Large integers have no size limitation.

Large integers are intended for arbitrary length integers. Regular integers types (positive and negative) are limited to 32-bit values.

¹⁸<http://twistedmatrix.com/products/reality>

¹⁹<http://nevow.com/>

Examples

Here are some examples of elements and their encodings - the type bytes are marked in bold:

```

1 0x01 0x81
-1 0x01 0x83

1.5 0x84 0x3f 0xf8 0x00 0x00 0x00 0x00 0x00 0x00

"hello" 0x05 0x82 0x68 0x65 0x6c 0x6c 0x6f

[] 0x00 0x80

[1, 23] 0x02 0x80 0x01 0x81 0x17 0x81

123456789123456789 0x15 0x3e 0x41 0x66 0x3a 0x69 0x26 0x5b 0x01 0x85

[1, ["hello"]] 0x02 0x80 0x01 0x81 0x01 0x80 0x05 0x82 0x68 0x65 0x6c 0x6c
                0x6f

```

9.3.4 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'

9.3.5 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.