An Introduction to Asynchronous Programming and Twisted

Dave Peticolas

Part 1: In Which We Begin at the Beginning

Preface

Someone recently posted to the Twisted mailing list asking for something like the "Twisted introduction for people on a deadline". Full disclosure: this isn't it. On the spectrum of introductions to Twisted and asynchronous programming in Python, it may be on the exact opposite end. So if you don't have any time, or any patience, this isn't the introduction you are looking for.

However, I also believe that if you are new to asynchronous programming, a quick introduction is simply not possible, at least if you are not a genius. I've used Twisted successfully for a number of years and having thought about how I initially learned it (slowly), and what I found difficult, I've come to the conclusion that much of the challenge does not stem from Twisted per se, but rather in the acquisition of the "mental model" required to write and understand asynchronous code. Most of the Twisted source code is clear and well written, and the online documentation is good, at least by the standards of most free software. But without that mental model, reading the Twisted codebase, or code that uses Twisted, or even much of the documentation, will result in confusion and headache.

So the first parts of this introduction are designed to help you acquire that model and only later on will we introduce the features of Twisted. In fact, we will start without using Twisted at all, instead using simple Python programs to illustrate how an asynchronous system works. And once we get into Twisted, we will begin with very low-level aspects that you would not normally use in day-to-day programming. Twisted is a highly abstracted system and this gives you tremendous leverage when you use it to solve problems. But when you are learning Twisted, and particularly when you are trying to understand how Twisted actually works, the many levels of abstraction can cause troubles. So we will go from the inside-out, starting with the basics.

And once you have the mental model in place, I think you will find reading the Twisted documentation, or just browsing the source code, to be much easier. So let's begin.

The Models

We will start by reviewing two (hopefully) familiar models in order to contrast them with the asynchronous model. By way of illustration we will imagine a program that consists of three conceptually distinct tasks which must be performed to complete the program. We will make these tasks more concrete later on, but for now we won't say anything about them except the program must perform them. Note I am using "task" in the non-technical sense of "something that needs to be done".

The first model we will look at is the single-threaded synchronous model, in Figure 1 below:

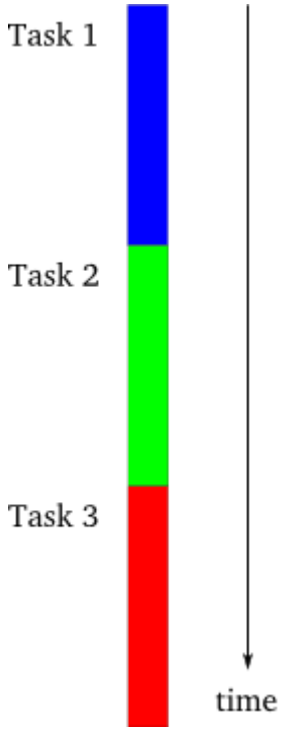

Figure 1: the synchronous model

This is the simplest style of programming. Each task is perfomed one at a time, with one finishing completely before another is started. And if the tasks are always performed in a definite order, the implementation of a later task can assume that all earlier tasks have finished without errors, with all their output available for use — a definite simplification in logic.

We can contrast the synchronous model with another one, the threaded model illustrated in Figure 2:

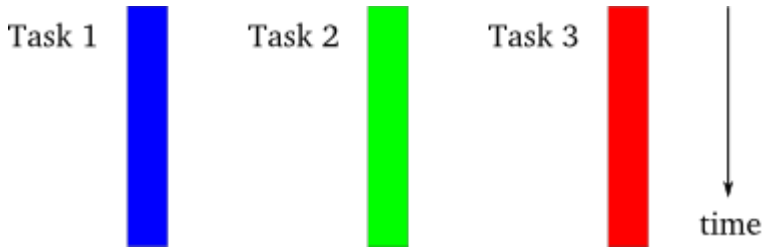

Figure 2: the threaded model

In this model, each task is performed in a separate thread of control. The threads are managed by the operating system and may, on a system with multiple processors or multiple cores, run truly concurrently, or may be interleaved together on a single processor. The point is, in the threaded model the details of execution are handled by the OS and the programmer simply thinks in terms of independent instruction streams which may run simultaneously. Although the diagram is simple, in practice threaded programs can be quite complex because of the need for threads to coordinate with one another. Thread communication and coordination is an advanced programming topic and can be difficult to get right.

Some programs implement parallelism using multiple processes instead of multiple threads. Although the programming details are different, for our purposes it is the same model as in Figure 2.

Now we can introduce the asynchronous model in Figure 3:

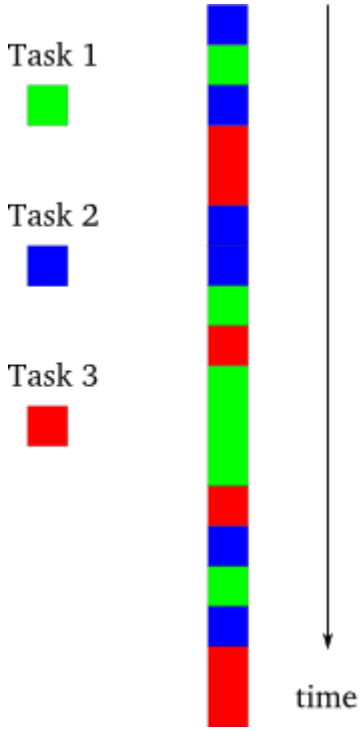

Figure 3: the asynchronous model

In this model, the tasks are interleaved with one another, but in a single thread of control. This is simpler than the threaded case because the programmer always knows that when one task is executing, another task is not. Although in a single-processor system a threaded program will also execute in an interleaved pattern, a programmer using threads should still think in terms of Figure 2, not Figure 3, lest the program work incorrectly when moved to a multi-processor system. But a single-threaded asynchronous system will always execute with interleaving, even on a multi-processor system.

There is another difference between the asynchronous and threaded models. In a threaded system the decision to suspend one thread and execute another is largely outside of the programmer's control. Rather, it is under the control of the operating system, and the programmer must assume that a thread may be suspended and replaced with another at almost any time. In contrast, under the asynchronous model a task will continue to run until it explicitly relinquishes control to other tasks. This is a further simplification from the threaded case.

> Note that it is possible to mix the asynchronous and threaded models and use both in the same system. But for most of this introduction, we will stick to "plain vanilla" asynchronous systems with one thread of control.

The Motivation

We've seen that the asynchronous model is simpler than the threaded one because there is a single instruction stream and tasks explicitly relinquish control instead of being suspended arbitrarily. But the asynchronous model is clearly more complex than the synchronous case. The programmer must organize each task as a sequence of smaller steps that execute intermittently. And if one task uses the output of another, the dependent task must be written to accept its input as a series of bits and pieces instead of all together. Since there is no actual parallelism, it appears from our diagrams that an asynchronous program will take just as long to execute as a synchronous one, perhaps longer as the asynchronous program might exhibit poorer locality of reference.

So why would you choose to use the asynchronous model? There are at least two reasons. First, if one or more of the tasks are responsible for implementing an interface for a human being, then by interleaving

the tasks together the system can remain responsive to user input while still performing other work in the "background". So while the background tasks may not execute any faster, the system will be more pleasant for the person using it.

However, there is a condition under which an asynchronous system will simply outperform a synchronous one, sometimes dramatically so, in the sense of performing all of its tasks in an overall shorter time. This condition holds when tasks are forced to wait, or *block*, as illustrated in Figure 4:

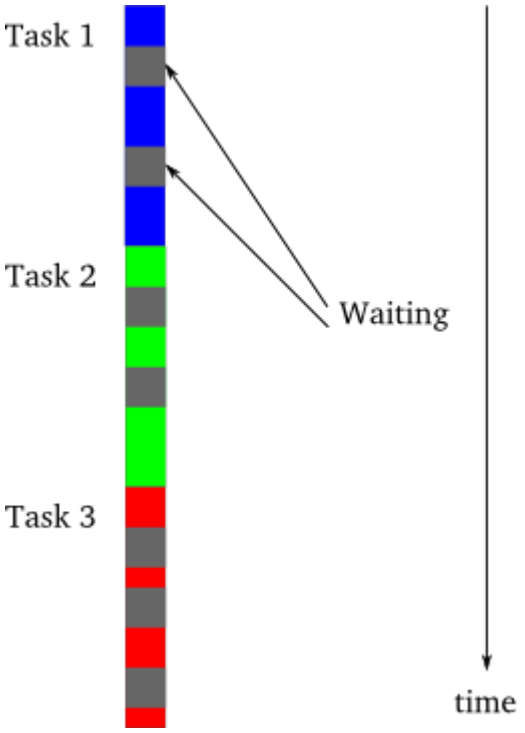

Figure 4: blocking in a synchronous program

In the figure, the gray sections represent periods of time when a particular task is waiting (blocking) and thus cannot make any progress. Why would a task be blocked? A frequent reason is that it is waiting to perform I/O, to transfer data to or from an external device. A typical CPU can handle data transfer rates that are orders of magnitude faster than a disk or a network link is capable of sustaining. Thus, a synchronous program that is doing lots of I/O will spend much of its time blocked while a disk or network catches up. Such a synchronous program is also called a blocking program for that reason.

Notice that Figure 4, a blocking program, looks a bit like Figure 3, an asynchronous program. This is not a coincidence. The fundamental idea behind the asynchronous model is that an asynchronous program, when faced with a task that would normally block in a synchronous program, will instead execute some other task that can still make progress. So an asynchronous program only "blocks" when no task can make progress and is thus called a non-blocking program. And each switch from one task to another corresponds to the first task either finishing, or coming to a point where it would have to block. With a large number of potentially blocking tasks, an asynchronous program can outperform a synchronous one by spending less overall time waiting, while devoting a roughly equal amount of time to real work on the individual tasks.

Compared to the synchronous model, the asynchronous model performs best when:

1. There are a large number of tasks so there is likely always at least one task that can make progress.
2. The tasks perform lots of I/O, causing a synchronous program to waste lots of time blocking when other tasks could be running.
3. The tasks are largely independent from one another so there is little need for inter-task communication (and thus for one task to wait upon another).

These conditions almost perfectly characterize a typical busy network server (like a web server) in a client-server environment. Each task represents one client request with I/O in the form of receiving the request and sending the reply. And client requests (being mostly reads) are largely independent. So a network server implementation is a prime candidate for the asynchronous model and this is why Twisted is first and foremost a networking library.

Onward and Upward

This is the end of Part 1. In Part 2, we will write some network programs, both blocking and non-blocking, as simply as possible (without using Twisted), to get a feel for how an asynchronous Python program actually works.

Part 2: Slow Poetry and the Apocalypse

This continues the introduction started here. And if you read it, welcome back. Now we're going to get our hands dirty and write some code. But first, let's get some assumptions out of the way.

My Assumptions About You

I will proceed as if you have a basic working knowledge of writing synchronous programs in Python, and know at least a little bit about Python socket programming. If you have never used sockets before, you might read the socket module documentation now, especially the example code towards the end. If you've never used Python before, then the rest of this introduction is probably going to be rather opaque.

My Assumptions About Your Computer

My experience with Twisted is mainly on Linux systems, and it is a Linux system on which I developed the examples. And while I won't intentionally make the code Linux-dependent, some of it, and some of what I say, may only apply to Linux and other UNIX-like systems (like Mac OSX or FreeBSD). Windows is a strange, murky place and, if you are hacking in it, I can't offer you much more beyond my heartfelt sympathies.

I will assume you have installed relatively recent versions of Python and Twisted. The examples were developed with Python 2.5 and Twisted 8.2.0.

Also, you can run all the examples on a single computer, although you can configure them to run on a network of systems as well. But for learning the basic mechanics of asynchronous programming, a single computer will do fine.

Getting the example code

The example code is available as a zip or tar file or as a clone of my public git repository. If you can use git or another version control system that can read git repositories, then I recommend using that method as I will update the examples over time and it will be easier for you to stay current. As a bonus, it includes the SVG source files used to generate the figures. Here is the git command to clone the repository:

```
git clone git://github.com/jdavisp3/twisted-intro.git
```

The rest of this tutorial will assume you have the latest copy of the example code and you have multiple shells open in its top-level directory (the one with the README file).

Slow Poetry

Although CPUs are much faster than networks, most networks are still a lot faster than your brain, or at

least faster than your eyeballs. So it can be challenging to get the "cpu's-eye-view" of network latency, especially when there's only one machine and the bytes are whizzing past at full speed on the loopback interface. What we need is a slow server, one with artificial delays we can vary to see the effect. And since servers have to serve something, ours will serve poetry. The example code includes a sub-directory called `poetry` with one poem each by John Donne, W.B. Yeats, and Edgar Allen Poe. Of course, you are free to substitute your own poems for the server to dish up.

The basic slow poetry server is implemented in blocking-server/slowpoetry.py. You can run one instance of the server like this:

```
python blocking-server/slowpoetry.py poetry/ecstasy.txt
```

That command will start up the blocking server with John Donne's poem "Ecstasy" as the poem to serve. Go ahead and look at the source code to the blocking server now. As you can see, it does not use Twisted, only basic Python socket operations. It also sends a limited number of bytes at a time, with a fixed time delay between them. By default, it sends 10 bytes every 0.1 seconds, but you can change these parameters with the –num-bytes and –delay command line options. For example, to send 50 bytes every 5 seconds:

```
python blocking-server/slowpoetry.py --num-bytes 50 --delay 5 poetry/ecstasy.txt
```

When the server starts up it prints out the port number it is listening on. By default, this is a random port that happens to be available on your machine. When you start varying the settings, you will probably want to use the same port number over again so you don't have to adjust the client command. You can specify a particular port like this:

```
python blocking-server/slowpoetry.py --port 10000 poetry/ecstasy.txt
```

If you have the netcat program available, you could test the above command like this:

```
netcat localhost 10000
```

If the server is working, you will see the poem slowly crawl its way down your screen. Ecstasy! You will also notice the server prints out a line each time it sends some bytes. Once the complete poem has been sent, the server closes the connection.

By default, the server only listens on the local "loopback" interface. If you want to access the server from another machine, you can specify the interface to listen on with the –iface option.

Not only does the server send each poem slowly, if you read the code you will find that while the server is sending poetry to one client, all other clients must wait for it to finish before getting even the first line. It is truly a slow server, and not much use except as a learning device.

> **Or is it?**
>
> On the other hand, if the more pessimistic of the Peak Oil folks are right and our world is heading for a global energy crisis and planet-wide societal meltdown, then perhaps one day soon a low-bandwidth, low-power poetry server could be just what we need. Imagine, after a long day of tending your self-sufficient gardens, making your own clothing, serving on your commune's Central Organizing Committee, and fighting off the radioactive zombies that roam the post-apocalyptic wastelands, you could crank up your generator and download a few lines of high culture from a vanished civilization. That's when our little server will really come into its own.

The Blocking Client

Also in the example code is a blocking client which can download poems from multiple servers, one after another. Let's give our client three tasks to perform, as in Figure 1 from Part 1. First we'll start three

servers, serving three different poems. Run these commands in three different terminal windows:

```
python blocking-server/slowpoetry.py --port 10000 poetry/ecstasy.txt --num-bytes 30
python blocking-server/slowpoetry.py --port 10001 poetry/fascination.txt
python blocking-server/slowpoetry.py --port 10002 poetry/science.txt
```

You can choose different port numbers if one or more of the ones I chose above are already being used on your system. Note I told the first server to use chunks of 30 bytes instead of the default 10 since that poem is about three times as long as the others. That way they all finish around the same time.

Now we can use the blocking client in `blocking-client/get-poetry.py` to grab some poetry. Run the client like this:

```
python blocking-client/get-poetry.py 10000 10001 10002
```

Change the port numbers here, too, if you used different ones for your servers. Since this is the blocking client, it will download one poem from each port number in turn, waiting until a complete poem is received until starting the next. Instead of printing out the poems, the blocking client produces output like this:

```
Task 1: get poetry from: 127.0.0.1:10000
Task 1: got 3003 bytes of poetry from 127.0.0.1:10000 in 0:00:10.126361
Task 2: get poetry from: 127.0.0.1:10001
Task 2: got 623 bytes of poetry from 127.0.0.1:10001 in 0:00:06.321777
Task 3: get poetry from: 127.0.0.1:10002
Task 3: got 653 bytes of poetry from 127.0.0.1:10002 in 0:00:06.617523
Got 3 poems in 0:00:23.065661
```

This is basically a text version of Figure 1, where each task is downloading a single poem. Your times may be a little different, and will vary as you change the timing parameters of the servers. Try changing those parameters to see the effect on the download times.

You might take a look at the source code to the blocking server and client now, and locate the points in the code where each blocks while sending or receiving network data.

The Asynchronous Client

Now let's take a look at a simple asynchronous client written without Twisted. First let's run it. Get a set of three servers going on the same ports like we did above. If the ones you ran earlier are still going, you can just use them again. Now we can run the asynchronous client, located in `async-client/get-poetry.py`, like this:

```
python async-client/get-poetry.py 10000 10001 10002
```

And you should get some output like this:

```
Task 1: got 30 bytes of poetry from 127.0.0.1:10000
Task 2: got 10 bytes of poetry from 127.0.0.1:10001
Task 3: got 10 bytes of poetry from 127.0.0.1:10002
Task 1: got 30 bytes of poetry from 127.0.0.1:10000
Task 2: got 10 bytes of poetry from 127.0.0.1:10001
...
Task 1: 3003 bytes of poetry
Task 2: 623 bytes of poetry
Task 3: 653 bytes of poetry
Got 3 poems in 0:00:10.133169
```

This time the output is much longer because the asynchronous client prints a line each time it downloads some bytes from any server, and these slow poetry servers just dribble out the bytes little by little. Notice that the individual tasks are mixed together just like in Figure 3 from Part 1.

Try varying the delay settings for the servers (e.g., by making one server slower than the others) to see how the asynchronous client automatically "adjusts" to the speed of the slower servers while still keeping up with the faster ones. That's asynchronicity in action.

Also notice that, for the server settings we chose above, the asynchronous client finishes in about 10 seconds while the synchronous client needs around 23 seconds to get all the poems. Now recall the differences between Figure 3 and Figure 4 in Part 1. By spending less time blocking, our asynchronous client can download all the poems in a shorter overall time. Now, our asynchronous client does block some of the time. Our slow server is *slow*. It's just that the asynchronous client spends a lot less time blocking than the "blocking" client does, because it can switch back and forth between all the servers.

> Technically, our asynchronous client *is* performing a blocking operation: it's writing to the standard output file descriptor with those `print` statements! This isn't a problem for our examples. On a local machine with a terminal shell that's always willing to accept more output the `print` statements won't really block, and execute quickly relative to our slow servers. But if we wanted our program to be part of a process pipeline and still execute asynchronously, we would need to use asynchronous I/O for standard input and output, too. Twisted includes support for doing just that, but to keep things simple we're just going to use `print` statements, even in our Twisted programs.

A Closer Look

Now take a look at the source code for the asynchronous client. Notice the main differences between it and the synchronous client:

1. Instead of connecting to one server at a time, the asynchronous client connects to all the servers at once.
2. The socket objects used for communication are placed in non-blocking mode with the call to `setblocking(0)`.
3. The `select` method in the select module is used to wait (block) until any of the sockets are ready to give us some data.
4. When reading data from the servers, we read only as much as we can until the socket would block, and then move on to the next socket with data to read (if any). This means we have to keep track of the poetry we've received from each server so far.

The core of the asynchronous client is the top-level loop in the `get_poetry` function. This loop can be broken down into steps:

1. Wait (block) on all open sockets using `select` until one (or more) sockets has data to be read.
2. For each socket with data to be read, read it, but only as much as is available now. Don't block.
3. Repeat, until all sockets have been closed.

The synchronous client had a loop as well (in the `main` function), but each iteration of the synchronous loop downloaded one complete poem. In one iteration of the asynchronous client we might download pieces of all the poems we are working on, or just some of them. And we don't know which ones we will work on in a given iteration, or how much data we will get from each one. That all depends on the relative speeds of the servers and the state of the network. We just let `select` tell us which ones are ready to go, and then read as much data as we can from each socket without blocking.

If the synchronous client always contacted a fixed number of servers (say 3), it wouldn't need an outer loop at all, it could just call its blocking `get_poetry` function three times in succession. But the asynchronous client can't do without an outer loop — to gain the benefits of asynchronicity, we need to wait on *all* of our sockets at once, and only process as much data as each is capable of delivering in any given iteration.

This use of a loop which waits for events to happen, and then handles them, is so common that it has

achieved the status of a design pattern: the reactor pattern. It is visualized in Figure 5 below:

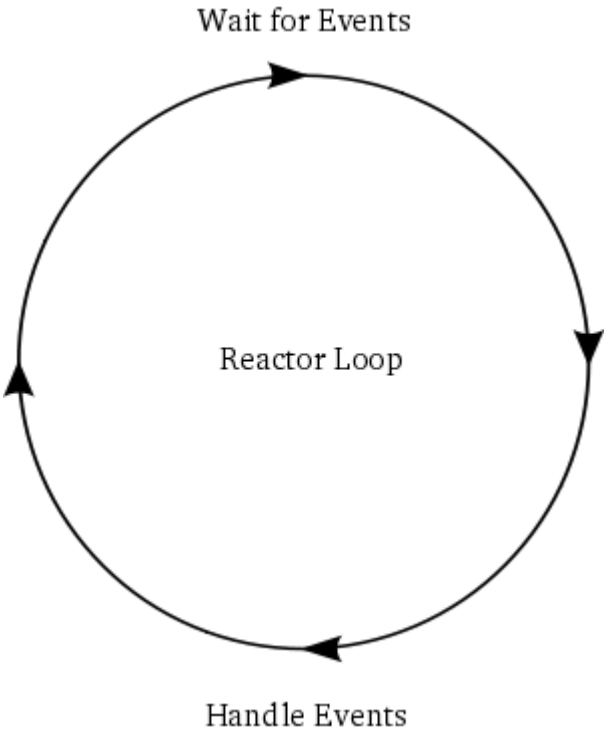

Figure 5: the reactor loop

The loop is a "reactor" because it waits for and then reacts to events. For that reason it is also known as an *event loop*. And since reactive systems are often waiting on I/O, these loops are also sometimes called *select loops*, since the select call is used to wait for I/O. So in a `select` loop, an "event" is when a socket becomes available for reading or writing. Note that `select` is not the only way to wait for I/O, it is just one of the oldest methods (and thus widely available). There are several newer APIs, available on different operating systems, that do the same thing as `select` but offer (hopefully) better performance. But leaving aside performance, they all do the same thing: take a set of sockets (really file descriptors) and block until one or more of them is ready to do I/O.

> Note that it's possible to use `select` and its brethren to simply check whether a set of file descriptors is ready for I/O without blocking. This feature permits a reactive system to perform non-I/O work inside the loop. But in reactive systems it is often the case that all work is I/O-bound, and thus blocking on all file descriptors conserves CPU resources.

Strictly speaking, the loop in our asynchronous client is not the reactor pattern because the loop logic is not implemented separately from the "business logic" that is specific to the poetry servers. They are all just mixed together. A real implementation of the reactor pattern would implement the loop as a separate abstraction with the ability to:

1. Accept a set of file descriptors you are interested in performing I/O with.
2. Tell you, repeatedly, when any file descriptors are ready for I/O.

And a really good implementation of the reactor pattern would also:

1. Handle all the weird corner cases that crop up on different systems.
2. Provide lots of nice abstractions to help you use the reactor with the least amount of effort.
3. Provide implementations of public protocols that you can use out of the box.

Well that's just what Twisted is — a robust, cross-platform implementation of the Reactor Pattern with lots of extras. And in Part 3 we will start writing some simple Twisted programs as we move towards a Twisted version of Get Poetry Now!.

Suggested Exercises

1. Do some timing experiments with the blocking and asynchronous clients by varying the number and settings of the poetry servers.
2. Could the asynchronous client provide a `get_poetry` function that returned the text of the poem? Why not?
3. If you wanted a `get_poetry` function in the asynchronous client that was analogous to the synchronous version of `get_poetry`, how could it work? What arguments and return values might it have?

Part 3: Our Eye-beams Begin to Twist

This continues the introduction started here. You can find an index to the entire series here.

Doing Nothing, the Twisted Way

Eventually we are going to re-implement our asynchronous poetry client using Twisted. But first let's write a few really simple Twisted programs just to get the flavor of things. As I mentioned in Part 2, I developed these examples using Twisted 8.2.0. Twisted APIs do change, but the core APIs we are going to use will likely change slowly, if at all, so I expect these examples to work for many future releases. If you don't have Twisted installed you can obtain it here.

The absolute simplest Twisted program is listed below, and is also available in `basic-twisted/simple.py` in the base directory of the `twisted-intro` example code.

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

You can run it like this:

```
1 python basic-twisted/simple.py
```

As we saw in Part 2, Twisted is an implementation of the Reactor Pattern and thus contains an object that represents the reactor, or event loop, that is the heart of any Twisted program. The first line of our program imports the reactor object so we can use it, and the second line tells the reactor to start running the loop.

This program just sits there doing nothing. You'll have to stop it by pressing `Control-C`, otherwise it will just sit there forever. Normally we would have given the loop one or more file descriptors (connected to, say, a poetry server) that we want to monitor for I/O. We'll see how to do that later, but for now our reactor loop is stuck. Note that this is not a busy loop which keeps cycling over and over. If you happen to have a CPU meter on your screen, you won't see any spikes caused by this technically infinite loop. In fact, our program isn't using any CPU at all. Instead, the reactor is stuck at the top cycle of Figure 5, waiting for an event that will never come (to be specific, waiting on a `select` call with no file descriptors).

That might make for a compelling metaphor of Hamletian inaction, but it's still a pretty boring program. We're about to make it more interesting, but we can already draw a few conclusions:

1. Twisted's reactor loop doesn't start until told to. You start it by calling `reactor.run()`.
2. The reactor loop runs in the same thread it was started in. In this case, it runs in the main (and only) thread.
3. Once the loop starts up, it just keeps going. The reactor is now "in control" of the program (or the specific thread it was started in).
4. If it doesn't have anything to do, the reactor loop does not consume CPU.

5. The reactor isn't created explicitly, just imported.

That last point is worth elaborating on. In Twisted, the reactor is basically a Singleton. There is only one reactor object and it is created implicitly when you import it. If you open the `reactor` module in the `twisted.internet` package you will find very little code. The actual implementation resides in other files (starting with `twisted.internet.selectreactor`).

Twisted actually contains multiple reactor implementations. As mentioned in Part 2, the `select` call is just one method of waiting on file descriptors. It is the default method that Twisted uses, but Twisted does include other reactors that use other methods. For example, `twisted.internet.pollreactor` uses the `poll` system call instead of `select`.

To use an alternate reactor, you must `install` it *before* importing `twisted.internet.reactor`. Here is how you install the `pollreactor`:

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

If you import `twisted.internet.reactor` without first installing a specific reactor implementation, then Twisted will install the `selectreactor` for you. For that reason, it is general practice not to import the reactor at the top level of modules to avoid accidentally installing the default reactor. Instead, import the reactor in the same scope in which you use it.

> Note: as of this writing, Twisted has been moving gradually towards an architecture which would allow multiple reactors to co-exist. In this scheme, a reactor object would be passed around as a reference rather than imported from a module.
>
> Note: not all operating systems support the `poll` call. If that is the case for your system, this example will not work.

Now we can re-implement our first Twisted program using the `pollreactor`, as found in `basic-twisted/simple-poll.py`:

```
1 from twisted.internet import pollreactor
2 pollreactor.install()
3
4 from twisted.internet import reactor
5 reactor.run()
```

And we have a poll loop that does nothing at all instead of a select loop that does nothing at all. Neato.

We're going to stick with the default reactor for the rest of this introduction. For the purposes of learning Twisted, all the reactors do the same thing.

Hello, Twisted

Let's make a Twisted program that at least does *something*. Here's one that prints a message to the terminal window, after the reactor loop starts up:

```
1  def hello():
2      print 'Hello from the reactor loop!'
3      print 'Lately I feel like I\'m stuck in a rut.'
4
5  from twisted.internet import reactor
6
7  reactor.callWhenRunning(hello)
8
9  print 'Starting the reactor.'
10 reactor.run()
```

This program is in basic-twisted/hello.py. If you run it, you will see this output:

```
Starting the reactor.
Hello from the reactor loop!
Lately I feel like I'm stuck in a rut.
```

You'll still have to kill the program yourself, since it gets stuck again after printing those lines.

Notice the `hello` function is called after the reactor starts running. That means it is called by the reactor itself, so Twisted code must be calling our function. We arrange for this to happen by invoking the reactor method `callWhenRunning` with a reference to the function we want Twisted to call. And, of course, we have to do that before we start the reactor.

We use the term *callback* to describe the reference to the `hello` function. A callback is a function reference that we give to Twisted (or any other framework) that Twisted will use to "call us back" at the appropriate time, in this case right after the reactor loop starts up. Since Twisted's loop is separate from our code, most interactions between the reactor core and our business logic will begin with a callback to a function we gave to Twisted using various APIs.

We can see how Twisted is calling our code using this program:

```
1  import traceback
2
3  def stack():
4      print 'The python stack:'
5      traceback.print_stack()
6
7  from twisted.internet import reactor
8  reactor.callWhenRunning(stack)
9  reactor.run()
```

You can find it in basic-twisted/stack.py and it prints out something like this:

```
The python stack:
...
  reactor.run() <-- This is where we called the reactor
...
...  <-- A bunch of Twisted function calls
...
  traceback.print_stack() <-- The second line in the stack function
```

Don't worry about all the Twisted calls in between. Just notice the relationship between the `reactor.run()` call and our callback.

What's the deal with callbacks?

Twisted is not the only reactor framework that uses callbacks. The older asynchronous Python frameworks Medusa and asyncore also use them. As do the GUI toolkits GTK and QT, both based, like many GUI frameworks, on a reactor loop.

The developers of reactive systems sure love callbacks. Maybe they should just marry them. Maybe they already did. But consider this:

1. The reactor pattern is single-threaded.
2. A reactive framework like Twisted implements the reactor loop so our code doesn't have to.
3. Our code still needs to get called to implement our business logic.
4. Since it is "in control" of the single thread, the reactor loop will have to call our code.
5. The reactor can't know in advance which part of our code needs to be called.

In this situation callbacks are not just one option — they are the only real game in town.

Figure 6 shows what happens during a callback:

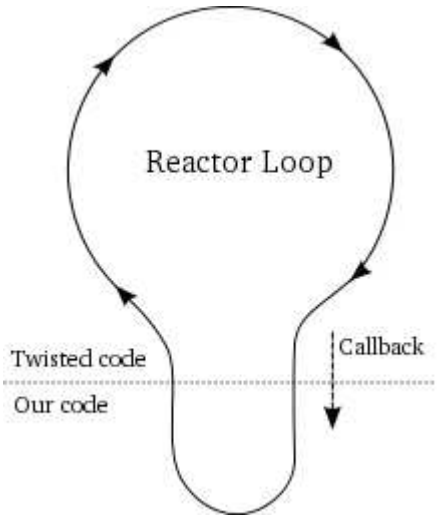

Figure 6: the reactor making a callback

Figure 6 illustrates some important properties of callbacks:

1. Our callback code runs in the same thread as the Twisted loop.
2. When our callbacks are running, the Twisted loop is not running.
3. And vice versa.
4. The reactor loop resumes when our callback returns.

During a callback, the Twisted loop is effectively "blocked" on our code. So we should make sure our callback code doesn't waste any time. In particular, we should avoid making blocking I/O calls in our callbacks. Otherwise, we would be defeating the whole point of using the reactor pattern in the first place. Twisted will not take any special precautions to prevent our code from blocking, we just have to make sure not to do it. As we will eventually see, for the common case of network I/O we don't have to worry about it as we let Twisted do the asynchronous communication for us.

Other examples of potentially blocking operations include reading or writing from a non-socket file descriptor (like a pipe) or waiting for a subprocess to finish. Exactly how you switch from blocking to non-blocking operations is specific to what you are doing, but there is often a Twisted API that will help you do it. Note that many standard Python functions have no way to switch to a non-blocking mode. For example, the `os.system` function will always block until the subprocess is finished. That's just how it works. So when using Twisted, you will have to eschew `os.system` in favor of the Twisted API for launching subprocesses.

Goodbye, Twisted

It turns out you can tell the Twisted reactor to stop running by using the reactor's `stop` method. But once stopped the reactor cannot be restarted, so it's generally something you do only when your program needs to exit.

> Note: there has been past discussion on the Twisted mailing list about making the reactor "restartable" so it could be started and stopped as you like. But as of version 8.2.0, you can only start (and thus stop) the reactor once.

Here's a program, listed in basic-twisted/countdown.py, which stops the reactor after a 5 second countdown:

```
1 | class Countdown(object):
```

```
    counter = 5

    def count(self):
        if self.counter == 0:
            reactor.stop()
        else:
            print self.counter, '...'
            self.counter -= 1
            reactor.callLater(1, self.count)

from twisted.internet import reactor

reactor.callWhenRunning(Countdown().count)

print 'Start!'
reactor.run()
print 'Stop!'
```

This program uses the `callLater` API to register a callback with Twisted. With `callLater` the callback is the second argument and the first argument is the number of seconds in the future you would like your callback to run. You can use a floating point number to specify a fractional number of seconds, too.

So how does Twisted arrange to execute the callback at the right time? Since this program doesn't listen on any file descriptors, why doesn't it get stuck in the `select` loop like the others? The `select` call, and the others like it, also accepts an optional *timeout* value. If a timeout value is supplied and no file descriptors have become ready for I/O within the specified time then the `select` call will return anyway. Incidentally, by passing a timeout value of zero you can quickly check (or "poll") a set of file descriptors without blocking at all.

You can think of a timeout as another kind of event the event loop of Figure 5 is waiting for. And Twisted uses timeouts to make sure any "timed callbacks" registered with `callLater` get called at the right time. Or rather, at approximately the right time. If another callback takes a really long time to execute, a timed callback may be delayed past its schedule. Twisted's `callLater` mechanism cannot provide the sort of guarantees required in a hard real-time system.

Here is the output of our countdown program:

```
Start!
5 ...
4 ...
3 ...
2 ...
1 ...
Stop!
```

Note the "Stop!" line at the ends shows us that when the reactor exits, the `reactor.run` call returns. And we have a program that stops all by itself.

Take That, Twisted

Since Twisted often ends up calling our code in the form of callbacks, you might wonder what happens when a callback raises an exception. Let's try it out. The program in `basic-twisted/exception.py` raises an exception in one callback, but behaves normally in another:

```
def falldown():
    raise Exception('I fall down.')

def upagain():
    print 'But I get up again.'
    reactor.stop()
```

```
 7
 8  from twisted.internet import reactor
 9
10  reactor.callWhenRunning(falldown)
11  reactor.callWhenRunning(upagain)
12
13  print 'Starting the reactor.'
14  reactor.run()
```

When you run it at the command line, you will see this output:

```
Starting the reactor.
Traceback (most recent call last):
  ... # I removed most of the traceback
exceptions.Exception: I fall down.
But I get up again.
```

Notice the second callback runs after the first, even though we see the traceback from the exception the first raised. And if you comment out the `reactor.stop()` call, the program will just keep running forever. So the reactor will keep going even when our callbacks fail (though it will report the exception).

Network servers generally need to be pretty robust pieces of software. They're not supposed to crash whenever any random bug shows its head. That's not to say we should be lackadaisical when it comes to handling our own errors, but it's nice to know Twisted has our back.

Poetry, Please

Now we're ready to grab some poetry with Twisted. In Part 4, we will implement a Twisted version of our asynchronous poetry client.

Suggested Exercises

1. Update the `countdown.py` program to have three independently running counters going at different rates. Stop the reactor when all counters have finished.
2. Consider the `LoopingCall` class in `twisted.internet.task`. Rewrite the countdown program above to use `LoopingCall`. You only need the `start` and `stop` methods and you don't need to use the "deferred" return value in any way. We'll learn what a "deferred" value is in a later Part.

Part 4: Twisted Poetry

This continues the introduction started here. You can find an index to the entire series here.

Our First Twisted Client

Although Twisted is probably more often used to write servers, clients are simpler than servers and we're starting out as simply as possible. Let's try out our first poetry client written with Twisted. The source code is in `twisted-client-1/get-poetry.py`. Start up some poetry servers as before:

```
python blocking-server/slowpoetry.py --port 10000 poetry/ecstasy.txt --num-bytes 30
python blocking-server/slowpoetry.py --port 10001 poetry/fascination.txt
python blocking-server/slowpoetry.py --port 10002 poetry/science.txt
```

And then run the client like this:

```
python twisted-client-1/get-poetry.py 10000 10001 10002
```

And you should get some output like this:

```
Task 1: got 60 bytes of poetry from 127.0.0.1:10000
Task 2: got 10 bytes of poetry from 127.0.0.1:10001
Task 3: got 10 bytes of poetry from 127.0.0.1:10002
Task 1: got 30 bytes of poetry from 127.0.0.1:10000
Task 3: got 10 bytes of poetry from 127.0.0.1:10002
Task 2: got 10 bytes of poetry from 127.0.0.1:10001
...
Task 1: 3003 bytes of poetry
Task 2: 623 bytes of poetry
Task 3: 653 bytes of poetry
Got 3 poems in 0:00:10.134220
```

Just like we did with our non-Twisted asynchronous client. Which isn't surprising as they are doing essentially the same thing. Let's take a look at the source code to see how it works. Open up the client in your editor so you can examine the code we are discussing.

> **Note**: As I mentioned in Part 1, we will begin our use of Twisted by using some very low-level APIs. By doing this we bypass some of the layers of Twisted's abstractions so we can learn Twisted from the "inside out". But this means a lot of the APIs we will learn in the beginning are not often used when writing real code. Just keep in mind that these early programs are learning exercises, not examples of how to write production software.

The Twisted client starts up by creating a set of `PoetrySocket` objects. A `PoetrySocket` initializes itself by creating a real network socket, connecting to a server, and switching to non-blocking mode:

```
1  self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
2  self.sock.connect(address)
3  self.sock.setblocking(0)
```

Eventually we'll get to a level of abstraction where we aren't working with sockets at all, but for now we still need to. After creating the network connection, a `PoetrySocket` passes *itself* to the `reactor` via the `addReader` method:

```
1  # tell the Twisted reactor to monitor this socket for reading
2  from twisted.internet import reactor
3  reactor.addReader(self)
```

This method gives Twisted a file descriptor you want to monitor for incoming data. Why are we passing Twisted an object instead of a file descriptor and a callback? And how will Twisted know what to do with our object since Twisted certainly doesn't contain any poetry-specific code? Trust me, I've looked. Open up the `twisted.internet.interfaces` module and follow along with me.

Twisted Interfaces

There are a number of sub-modules in Twisted called `interfaces`. Each one defines a set of `Interface` classes. As of version 8.0, Twisted uses `zope.interface` as the basis for those classes, but the details of that package aren't so important for us. We're just concerned with the Interface sub-classes in Twisted itself, like the ones you are looking at now.

One of the principle purposes of Interfaces is documentation. As a Python programmer you are doubtless familiar with Duck Typing, the notion that the type of an object is principally defined not by its position in a class hierarchy but by the public interface it presents to the world. Thus two objects which present the same public interface (i.e., walk like a duck, quack like a …) are, as far as duck typing is concerned, the same sort of thing (a duck!). Well an Interface is a somewhat formalized way of specifying just what it means to walk like a duck.

Skip down the `twisted.internet.interfaces` source code until you come to the definition of the `addReader` method. It is declared in the `IReactorFDSet` Interface and should look something like this:

```
def addReader(reader):
    """
    I add reader to the set of file descriptors to get read events for.

    @param reader: An L{IReadDescriptor} provider that will be checked for
                   read events until it is removed from the reactor with
                   L{removeReader}.

    @return: C{None}.
    """
```

`IReactorFDSet` is one of the Interfaces that Twisted reactors implement. Thus, any Twisted reactor has a method called `addReader` that works as described by the docstring above. The method declaration does not have a `self` argument because it is solely concerned with defining a public interface, and the `self` argument is part of the implementation (i.e., the caller does not have to pass `self` explicitly). Interface objects are never instantiated or used as base classes for real implementations.

> **Note 1:** Technically, `IReactorFDSet` would only be implemented by reactors that support waiting on file descriptors. As far as I know, that currently includes all available reactor implementations.
>
> **Note 2**: It is possible to use Interfaces for more than documentation. The `zope.interface` module allows you to explicitly declare that a class implements one or more interfaces, and provides mechanisms to examine these declarations at run-time. Also supported is the concept of adaptation, the ability to dynamically provide a given interface for an object that might not support that interface directly. But we're not going to delve into these more advanced use cases.
>
> **Note 3:** You might notice a similarity between Interfaces and Abstract Base Classes, a recent addition to the Python language. We will not be exploring their similarities and differences here, but you might be interested in reading an essay by Glyph, the Twisted project founder, that touches on that subject.

According to the docstring above, the `reader` argument of `addReader` should implement the `IReadDescriptor` interface. And that means our `PoetrySocket` objects have to do just that.

Scrolling through the module to find this new interface, we see:

```
class IReadDescriptor(IFileDescriptor):

    def doRead():
        """
        Some data is available for reading on your descriptor.
        """
```

And you will find an implementation of `doRead` on our `PoetrySocket` objects. It reads data from the socket asynchronously, whenever it is called by the Twisted reactor. So `doRead` is really a callback, but instead of passing it directly to Twisted, we pass in an object with a `doRead` method. This is a common idiom in the Twisted framework — instead of passing a function you pass an object that must implement a given Interface. This allows us to pass a set of related callbacks (the methods defined by the Interface) with a single argument. It also lets the callbacks communicate with each other through shared state stored on the object.

So what other callbacks are implemented on `PoetrySocket` objects? Notice that `IReadDescriptor` is a sub-class of `IFileDescriptor`. That means any object that implements `IReadDescriptor` must also implement `IFileDescriptor`. And if you do some more scrolling, you will find:

```
class IFileDescriptor(ILoggingContext):
    """
    A file descriptor.
    """

```

```
6       def fileno():
7           ...
8
9       def connectionLost(reason):
10          ...
```

I left out the docstrings above, but the purpose of these callbacks is fairly clear from the names: `fileno` should return the file descriptor we want to monitor, and `connectionLost` is called when the connection is closed. And you can see our `PoetrySocket` objects implement those methods as well.

Finally, `IFileDescriptor` inherits from `ILoggingContext`. I won't bother to show it here, but that's why we need to implement the `logPrefix` callback. You can find the details in the `interfaces` module.

> **Note**: You might notice that `doRead` is returning special values to indicate when the socket is closed. How did I know to do that? Basically, it didn't work without it and I peeked at Twisted's implementation of the same interface to see what to do. You may wish to sit down for this: sometimes software documentation is wrong or incomplete. Perhaps when you have recovered from the shock, I'll have finished Part 5.

More on Callbacks

Our new Twisted client is really quite similar to our original asynchronous client. Both clients connect their own sockets, and read data from those sockets (asynchronously). The main difference is the Twisted client doesn't need its own `select` loop — it uses the Twisted reactor instead.

The `doRead` callback is the most important one. Twisted calls it to tell us there is some data ready to read from our socket. We can visualize the process in Figure 7:

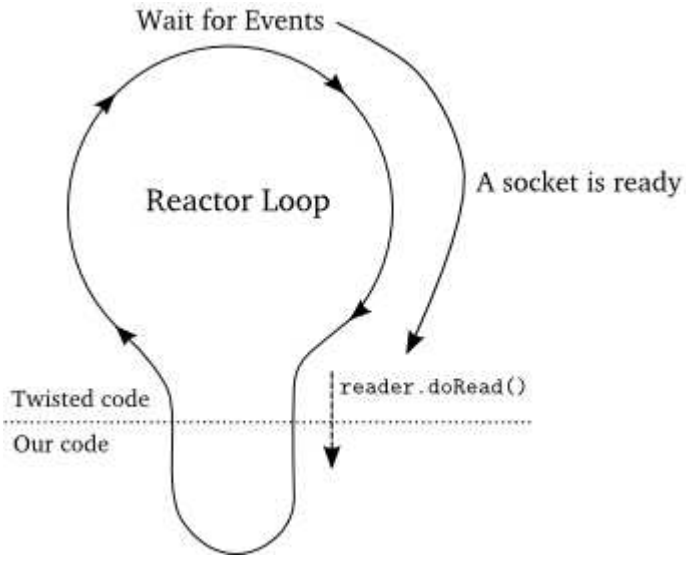

Figure 7: the doRead callback

Each time the callback is invoked it's up to us to read all the data we can and then stop without blocking. And as we said in Part 3, Twisted can't stop our code from misbehaving (from blocking needlessly). We can do just that and see what happens. In the same directory as our Twisted client is a broken client called `twisted-client-1/get-poetry-broken.py`. This client is identical to the one you've been looking at, with two exceptions:

1. The broken client doesn't bother to make the socket non-blocking.
2. The `doRead` callback just keeps reading bytes (and possibly blocking) until the socket is closed.

Now try running the broken client like this:

```
python twisted-client-1/get-poetry-broken.py 10000 10001 10002
```

You'll get some output that looks something like this:

```
Task 1: got 3003 bytes of poetry from 127.0.0.1:10000
Task 3: got 653 bytes of poetry from 127.0.0.1:10002
Task 2: got 623 bytes of poetry from 127.0.0.1:10001
Task 1: 3003 bytes of poetry
Task 2: 623 bytes of poetry
Task 3: 653 bytes of poetry
Got 3 poems in 0:00:10.132753
```

Aside from a slightly different task order this looks like our original blocking client. But that's because the broken client *is* a blocking client. By using a blocking `recv` call in our callback, we've turned our nominally asynchronous Twisted program into a synchronous one. So we've got the complexity of a `select` loop without any of the benefits of asynchronicity.

The sort of multi-tasking capability that an event loop like Twisted provides is cooperative. Twisted will tell us when it's OK to read or write to a file descriptor, but we have to play nice by only transferring as much data as we can without blocking. And we must avoid making other kinds of blocking calls, like `os.system`. Furthermore, if we have a long-running computational (CPU-bound) task, it's up to us to split it up into smaller chunks so that I/O tasks can still make progress if possible.

Note that there is a sense in which our broken client still works: it does manage to download all the poetry we asked it to. It's just that it can't take advantage of the efficiencies of asynchronous I/O. Now you might notice the broken client still runs a lot faster than the original blocking client. That's because the broken client connects to all the servers at the start of the program. Since the servers start sending data immediately, and since the OS will buffer some of the incoming data for us even if we don't read it (up to a limit), our blocking client is effectively receiving data from the other servers even though it is only reading from one at a time.

But this "trick" only works for small amounts of data, like our short poems. If we were downloading, say, the three 20 million-word epic sagas that chronicle one hacker's attempt to win his true love by writing the world's greatest Lisp interpreter, the operating system buffers would quickly fill up and our broken client would be scarcely more efficient than our original blocking one.

Wrapping Up

I don't have much more to say about our first Twisted poetry client. You might note the `connectionLost` callback shuts down the reactor after there are no more `PoetrySocket`s waiting for poems. That's not such a great technique since it assumes we aren't doing anything else in the program other than download poetry, but it does illustrate a couple more low-level reactor APIs, `removeReader` and `getReaders`.

There are `Writer` equivalents to the `Reader` APIs we used in this client, and they work in analogous ways for file descriptors we want to monitor for sending data to. Consult the `interfaces` file for more details. The reason reading and writing have separate APIs is because the `select` call distinguishes between those two kinds of events (a file descriptor becoming available for reading or writing, respectively). It is, of course, possible to wait for both events on the same file descriptor.

In Part 5, we will write a second version of our Twisted poetry client using some higher-level abstractions, and learn some more Twisted Interfaces and APIs along the way.

Suggested Exercises

1. Fix the client so that a failure to connect to a server does not crash the program.
2. Use `callLater` to make the client timeout if a poem hasn't finished after a given interval. Read about the return value of `callLater` so you can cancel the timeout if the poem finishes on time.

Part 5: Twistier Poetry

This continues the introduction started here. You can find an index to the entire series here.

Abstract Expressionism

In Part 4 we made our first poetry client that uses Twisted. It works pretty well, but there is definitely room for improvement.

First of all, the client includes code for mundane details like creating network sockets and receiving data from those sockets. Twisted provides support for these sorts of things so we don't have to implement them ourselves every time we write a new program. This is especially helpful because asynchronous I/O requires a few tricky bits involving exception handling as you can see in the client code. And there are even more tricky bits if you want your code to work on multiple platforms. If you have a free afternoon, search the Twisted sources for "win32" to see how many corner cases that platform introduces.

Another problem with the current client is error handling. Try running version 1.0 of the Twisted client and tell it to download from a port with no server. It just crashes. We could fix the current client, but error handling is easier with the Twisted APIs we'll be using today.

Finally, the client isn't particularly re-usable. How would another module get a poem with our client? How would the "calling" module know when the poem had finished downloading? We can't write a function that simply returns the text of the poem as that would require blocking until the entire poem is read. This is a real problem but we're not going to fix it today — we'll save that for future Parts.

We're going to fix the first and second problems using a higher-level set of APIs and Interfaces. The Twisted framework is loosely composed of layers of abstractions and learning Twisted means learning what those layers provide, i.e, what APIs, Interfaces, and implementations are available for use in each one. Since this is an introduction we're not going to study each abstraction in complete detail or do an exhaustive survey of every abstraction that Twisted offers. We're just going to look at the most important pieces to get a better feel for how Twisted is put together. Once you become familiar with the overall style of Twisted's architecture, learning new parts on your own will be much easier.

In general, each Twisted abstraction is concerned with one particular concept. For example, the 1.0 client from Part 4 uses `IReadDescriptor`, the abstraction of a "file descriptor you can read bytes from". A Twisted abstraction is usually defined by an Interface specifying how an object embodying that abstraction should behave. The most important thing to keep in mind when learning a new Twisted abstraction is this:

> Most higher-level abstractions in Twisted are built by *using* lower-level ones, *not* by replacing them.

So when you are learning a new Twisted abstraction, keep in mind both what it does and what it does not do. In particular, if some earlier abstraction *A* implements feature *F*, then *F* is probably not implemented by any other abstraction. Rather, if another abstraction *B* needs feature *F*, it will use *A* rather than implement *F* itself. (In general, an implementation of *B* will either sub-class an implementation of *A* or refer to another object that implements *A*).

Networking is a complex subject, and thus Twisted contains lots of abstractions. By starting with lower levels first, we are hopefully getting a clearer picture of how they all get put together in a working Twisted program.

Loopiness in the Brain

The most important abstraction we have learned so far, indeed the most important abstraction in Twisted, is the reactor. At the center of every program built with Twisted, no matter how many layers that program

might have, there is a reactor loop spinning around and making the whole thing go. Nothing else in Twisted provides the functionality the reactor offers. Much of the rest of Twisted, in fact, can be thought of as "stuff that makes it easier to do X using the reactor" where X might be "serve a web page" or "make a database query" or some other specific feature. Although it's possible to stick with the lower-level APIs, like the client 1.0 does, we have to implement more things ourselves if we do. Moving to higher-level abstractions generally means writing less code (and letting Twisted handle the platform-dependent corner cases).

But when we're working at the outer layers of Twisted it can be easy to forget the reactor is there. In any Twisted program of reasonable size, relatively few parts of our code will actually use the reactor APIs directly. The same is true for some of the other low-level abstractions. The file descriptor abstractions we used in client 1.0 are so thoroughly subsumed by higher-level concepts that they basically disappear in real Twisted programs (they are still used on the inside, we just don't see them as such).

As far as the file descriptor abstractions go, that's not really a problem. Letting Twisted handle the mechanics of asynchronous I/O frees us to concentrate on whatever problem we are trying to solve. But the reactor is different. It never really disappears. When you choose to use Twisted you are also choosing to use the Reactor Pattern, and that means programming in the "reactive style" using callbacks and cooperative multi-tasking. If you want to use Twisted correctly, you have to keep the reactor's existence (and the way it works) in mind. We'll have more to say about this in Part 6, but for now our message is this:

> Figure 5 and Figure 6 are the most important diagrams in this introduction.

We'll keep using diagrams to illustrate new concepts, but those two Figures are the ones that you need to burn into your brain, so to speak. Those are the pictures I constantly have in mind while writing programs with Twisted.

Before we dive into the code, there are three new abstractions to introduce: Transports, Protocols, and Protocol Factories.

Transports

The Transport abstraction is defined by `ITransport` in the main Twisted `interfaces` module. A Twisted Transport represents a single connection that can send and/or receive bytes. For our poetry clients, the Transports are abstracting TCP connections like the ones we have been making ourselves in earlier versions. But Twisted also supports I/O over UNIX Pipes and UDP sockets among other things. The Transport abstraction represents any such connection and handles the details of asynchronous I/O for whatever sort of connection it represents.

If you scan the methods defined for `ITransport`, you won't find any for receiving data. That's because Transports always handle the low-level details of reading data asynchronously from their connections, and give the data to us via callbacks. Along similar lines, the write-related methods of Transport objects may choose not to write the data immediately to avoid blocking. Telling a Transport to write some data means "send this data as soon as you can do so, subject to the requirement to avoid blocking". The data will be written in the order we provide it, of course.

We generally don't implement our own Transport objects or create them in our code. Rather, we use the implementations that Twisted already provides and which are created for us when we tell the reactor to make a connection.

Protocols

Twisted Protocols are defined by `IProtocol` in the same `interfaces` module. As you might expect, Protocol objects implement protocols. That is to say, a particular implementation of a Twisted Protocol

should implement one specific networking protocol, like FTP or IMAP or some nameless protocol we invent for our own purposes. Our poetry protocol, such as it is, simply sends all the bytes of the poem as soon as a connection is established, while the close of the connection signifies the end of the poem.

Strictly speaking, each instance of a Twisted Protocol object implements a protocol for one *specific* connection. So each connection our program makes (or, in the case of servers, accepts) will require one instance of a Protocol. This makes Protocol instances the natural place to store both the state of "stateful" protocols and the accumulated data of partially received messages (since we receive the bytes in arbitrary-sized chunks with asynchronous I/O).

So how do Protocol instances know what connection they are responsible for? If you look at the `IProtocol` definition, you will find a method called `makeConnection`. This method is a callback and Twisted code calls it with a Transport instance as the only argument. The Transport is the connection the Protocol is going to use.

Twisted includes a large number of ready-built Protocol implementations for various common protocols. You can find a few simpler ones in `twisted.protocols.basic`. It's a good idea to check the Twisted sources before you write a new Protocol to see if there's already an implementation you can use. But if there isn't, it's perfectly OK to implement your own, as we will do for our poetry clients.

Protocol Factories

So each connection needs its own Protocol and that Protocol might be an instance of a class we implement ourselves. Since we will let Twisted handle creating the connections, Twisted needs a way to make the appropriate Protocol "on demand" whenever a new connection is made. Making Protocol instances is the job of Protocol Factories.

As you've probably guessed, the Protocol Factory API is defined by IProtocolFactory, also in the `interfaces` module. Protocol Factories are an example of the Factory design pattern and they work in a straightforward way. The `buildProtocol` method is supposed to return a new Protocol instance each time it is called. This is the method that Twisted uses to make a new Protocol for each new connection.

Get Poetry 2.0: First Blood.0

Alright, let's take a look at version 2.0 of the Twisted poetry client. The code is in `twisted-client-2/get-poetry.py`. You can run it just like the others and get similar output so I won't bother posting output here. This is also the last version of the client that prints out task numbers as it receives bytes. By now it should be clear that all Twisted programs work by interleaving tasks and processing relatively small chunks of data at a time. We'll still use `print` statements to show what is going on at key moments, but the clients won't be quite as verbose in the future.

In client 2.0, sockets have disappeared. We don't even import the `socket` module and we never refer to a socket object, or a file descriptor, in any way. Instead, we tell the reactor to make the connections to the poetry servers on our behalf like this:

```
1  factory = PoetryClientFactory(len(addresses))
2
3  from twisted.internet import reactor
4
5  for address in addresses:
6      host, port = address
7      reactor.connectTCP(host, port, factory)
```

The `connectTCP` method is the one to focus on. The first two arguments should be self-explanatory. The third is an instance of our `PoetryClientFactory` class. This is the Protocol Factory for poetry clients and passing it to the reactor allows Twisted to create instances of our `PoetryProtocol` on demand.

Notice that we are not implementing either the Factory or the Protocol from scratch, unlike the `PoetrySocket` objects in our previous client. Instead, we are sub-classing the base implementations that Twisted provides in `twisted.internet.protocol`. The primary Factory base class is `twisted.internet.protocol.Factory`, but we are using the `ClientFactory` sub-class which is specialized for clients (processes that make connections instead of listening for connections like a server).

We are also taking advantage of the fact that the Twisted `Factory` class implements `buildProtocol` for us. We call the base class implementation in our sub-class:

```
1  def buildProtocol(self, address):
2      proto = ClientFactory.buildProtocol(self, address)
3      proto.task_num = self.task_num
4      self.task_num += 1
5      return proto
```

How does the base class know what Protocol to build? Notice we are also setting the class attribute `protocol` on `PoetryClientFactory`:

```
1  class PoetryClientFactory(ClientFactory):
2
3      task_num = 1
4
5      protocol = PoetryProtocol # tell base class what proto to build
```

The base `Factory` class implements `buildProtocol` by instantiating the class we set on `protocol` (`i.e.`, `PoetryProtocol`) and setting the `factory` attribute on that new instance to be a reference to its "parent" Factory. This is illustrated in Figure 8:

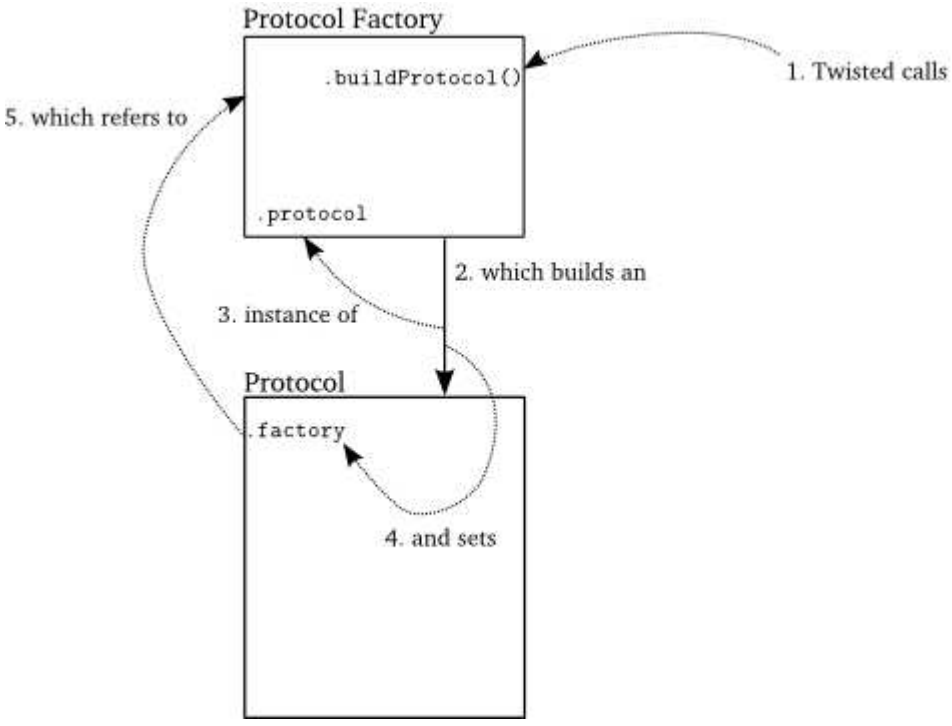

Figure 8: a Protocol is born

As we mentioned above, the `factory` attribute on Protocol objects allows Protocols created with the same Factory to share state. And since Factories are created by "user code", that same attribute allows Protocol objects to communicate results back to the code that initiated the request in the first place, as we will see in Part 6.

Note that while the `factory` attribute on Protocols refers to an instance of a Protocol Factory, the `protocol` attribute on the Factory refers to the *class* of the Protocol. In general, a single Factory might

create many Protocol instances.

The second stage of Protocol construction connects a Protocol with a Transport, using the `makeConnection` method. We don't have to implement this method ourselves since the Twisted base class provides a default implementation. By default, `makeConnection` stores a reference to the Transport on the `transport` attribute and sets the `connected` attribute to a True value, as depicted in Figure 9:

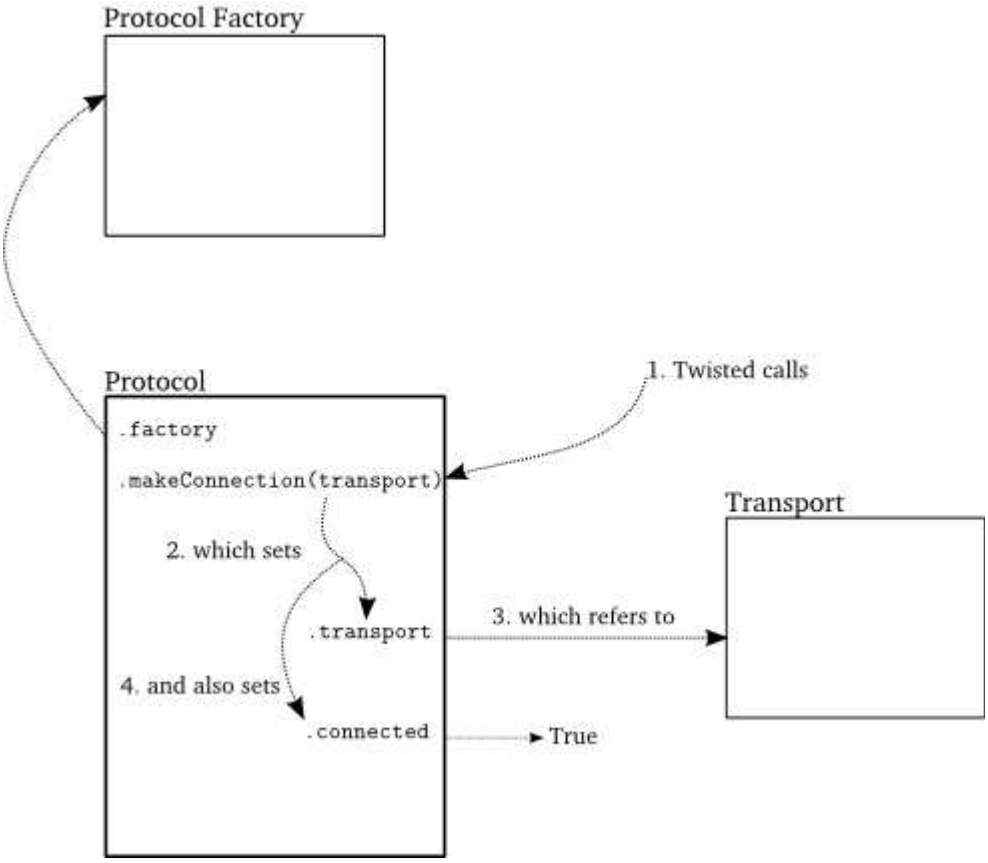

Figure 9: a Protocol meets its Transport

Once initialized in this way, the Protocol can start performing its real job — translating a lower-level stream of data into a higher-level stream of protocol messages (and vice-versa for 2-way connections). The key method for processing incoming data is `dataReceived`, which our client implements like this:

```
1  def dataReceived(self, data):
2      self.poem += data
3      msg = 'Task %d: got %d bytes of poetry from %s'
4      print  msg % (self.task_num, len(data), self.transport.getHost())
```

Each time `dataReceived` is called we get a new sequence of bytes (`data`) in the form of a string. As always with asynchronous I/O, we don't know how much data we are going to get so we have to buffer it until we receive a complete protocol message. In our case, the poem isn't finished until the connection is closed, so we just keep adding the bytes to our `.poem` attribute.

Note we are using the `getHost` method on our Transport to identify which server the data is coming from. We are only doing this to be consistent with earlier clients. Otherwise our code wouldn't need to use the Transport explicitly at all, since we never send any data to the servers.

Let's take a quick look at what's going on when the `dataReceived` method is called. In the same directory as our 2.0 client, there is another client called `twisted-client-2/get-poetry-stack.py`. This is just like the 2.0 client except the `dataReceived` method has been changed like this:

```
1  def dataReceived(self, data):
2      traceback.print_stack()
```

```
3       os._exit(0)
```

With this change the program will print a stack trace and then quit the first time it receives some data. You could run this version like so:

```
python twisted-client-2/get-poetry-stack.py 10000
```

And you will get a stack trace like this:

```
File "twisted-client-2/get-poetry-stack.py", line 125, in
    poetry_main()

... # I removed a bunch of lines here

File ".../twisted/internet/tcp.py", line 463, in doRead  # Note the doRead callback
    return self.protocol.dataReceived(data)
File "twisted-client-2/get-poetry-stack.py", line 58, in dataReceived
    traceback.print_stack()
```

There's the `doRead` callback we used in client 1.0! As we noted before, Twisted builds new abstractions by using the old ones, not by replacing them. So there is still an `IReadDescriptor` implementation hard at work, it's just implemented by Twisted instead of our code. If you are curious, Twisted's implementation is in `twisted.internet.tcp`. If you follow the code, you'll find that the same object implements `IWriteDescriptor` and `ITransport` too. So the `IReadDescriptor` is actually the Transport object in disguise. We can visualize a `dataReceived` callback with Figure 10:

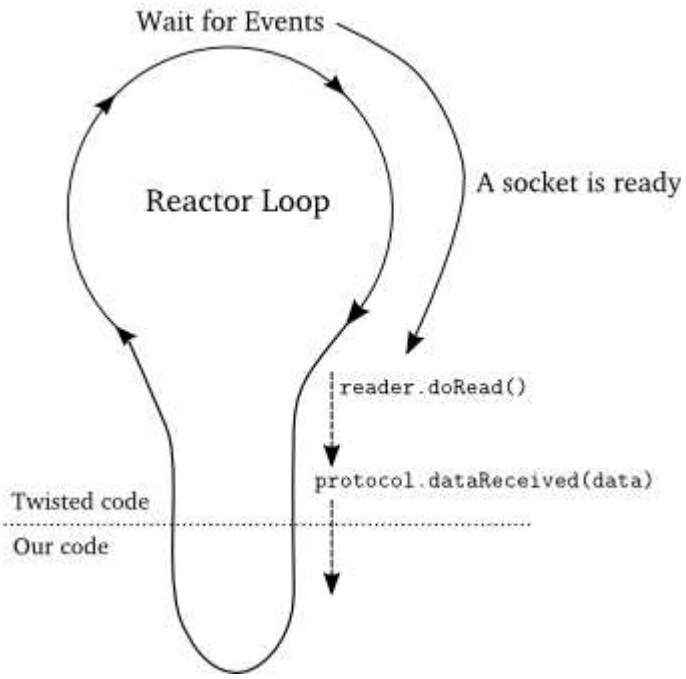

Figure 10: the dataReceived callback

Once a poem has finished downloading, the `PoetryProtocol` object notifies its `PoetryClientFactory`:

```
1   def connectionLost(self, reason):
2       self.poemReceived(self.poem)
3
4   def poemReceived(self, poem):
5       self.factory.poem_finished(self.task_num, poem)
```

The `connectionLost` callback is invoked when the transport's connection is closed. The `reason` argument is a `twisted.python.failure.Failure` object with additional information on whether the connection was closed cleanly or due to an error. Our client just ignores this value and assumes we received the entire poem.

The factory shuts down the reactor after all the poems are done. Once again we assume the only thing our program is doing is downloading poems, which makes `PoetryClientFactory` objects less reusable. We'll fix that in the next Part, but notice how the `poem_finished` callback keeps track of the number of poems left to go:

```
...
    self.poetry_count -= 1

    if self.poetry_count == 0:
        ...
```

If we were writing a multi-threaded program where each poem was downloaded in a separate thread we would need to protect this section of code with a lock in case two or more threads invoked `poem_finished` at the same time. Otherwise we might end up shutting down the reactor twice (and getting a traceback for our troubles). But with a reactive system we needn't bother. The reactor can only make one callback at a time, so this problem just can't happen.

Our new client also handles a failure to connect with more grace than the 1.0 client. Here's the callback on the `PoetryClientFactory` class which does the job:

```
def clientConnectionFailed(self, connector, reason):
    print 'Failed to connect to:', connector.getDestination()
    self.poem_finished()
```

Note the callback is on the factory, not on the protocol. Since a protocol is only created after a connection is made, the factory gets the news when a connection cannot be established.

A simpler client

Although our new client is pretty simple already, we can make it simpler if we dispense with the task numbers. The client should really be about the poetry, after all. There is a simplified 2.1 version in `twisted-client-2/get-poetry-simple.py`.

Wrapping Up

Client 2.0 uses Twisted abstractions that should be familiar to any Twisted hacker. And if all we wanted was a command-line client that printed out some poetry and then quit, we could even stop here and call our program done. But if we wanted some re-usable code, some code that we could embed in a larger program that needs to download some poetry but also do other things, then we still have some work to do. In Part 6 we'll take a first stab at it.

Suggested Exercises

1. Use `callLater` to make the client timeout if a poem hasn't finished after a given interval. Use the `loseConnection` method on the transport to close the connection on a timeout, and don't forget to cancel the timeout if the poem finishes on time.
2. Use the stacktrace method to analyze the callback sequence that occurs when `connectionLost` is invoked.

Part 6: And Then We Took It Higher

This continues the introduction started here. You can find an index to the entire series here.

Poetry for Everyone

We've made a lot of progress with our poetry client. Our last version (2.0) is using Transports, Protocols, and Protocol Factories, the workhorses of Twisted networking. But there are more improvements to make. Client 2.0 (and also 2.1) can only be used for downloading poetry at the command line. This is because the `PoetryClientFactory` is not only in charge of getting poetry, but also in charge of shutting down the program when it's finished. That's an odd job for something called "`PoetryClientFactory`", it really ought to do nothing beyond making `PoetryProtocol`s and collecting finished poems.

We need a way to send a poem to the code that requested the poem in the first place. In a synchronous program we might make an API like this:

```
1 def get_poetry(host, post):
2     """Return a poem from the poetry server at the given host and port."""
```

But of course, we can't do that here. The above function necessarily blocks until the poem is received in entirety, otherwise it couldn't work the way the documentation claims. But this is a reactive program so blocking on a network socket is out of the question. We need a way to tell the calling code when the poem is ready, without blocking while the poem is in transit. But this is the same sort of problem that Twisted itself has. Twisted needs to tell our code when a socket is ready for I/O, or when some data has been received, or when a timeout has occurred, etc. We've seen that Twisted solves this problem using callbacks, so we can use callbacks too:

```
1 def get_poetry(host, port, callback):
2     """
3     Download a poem from the given host and port and invoke
4 
5       callback(poem)
6 
7     when the poem is complete.
8     """
```

Now we have an asynchronous API we can use with Twisted, so let's go ahead and implement it.

> As I said before, we will at times be writing code in ways a typical Twisted programmer wouldn't. This is one of those times and one of those ways. We'll see in Parts 7 and 8 how to do this the "Twisted way" (surprise, it uses an abstraction!) but starting out simply will give us more insight into the finished version.

Client 3.0

You can find version 3.0 of our poetry client in `twisted-client-3/get-poetry.py`. This version has an implementation of the `get_poetry` function:

```
1 def get_poetry(host, port, callback):
2     from twisted.internet import reactor
3     factory = PoetryClientFactory(callback)
4     reactor.connectTCP(host, port, factory)
```

The only new wrinkle here is passing the callback function to the `PoetryClientFactory`. The factory uses the callback to deliver the poem:

```
1 class PoetryClientFactory(ClientFactory):
2 
3     protocol = PoetryProtocol
4 
5     def __init__(self, callback):
6         self.callback = callback
7 
8     def poem_finished(self, poem):
9         self.callback(poem)
```

Notice the factory is much simpler than in version 2.1 since it's no longer in charge of shutting the reactor down. It's also missing the code for detecting failures to connect, but we'll fix that in a little bit. The `PoetryProtocol` itself doesn't need to change at all so we just re-use the one from client 2.1:

```
class PoetryProtocol(Protocol):

    poem = ''

    def dataReceived(self, data):
        self.poem += data

    def connectionLost(self, reason):
        self.poemReceived(self.poem)

    def poemReceived(self, poem):
        self.factory.poem_finished(poem)
```

With this change, the `get_poetry` function, and the `PoetryClientFactory` and `PoetryProtocol` classes, are now completely re-usable. They are all about downloading poetry and nothing else. All the logic for starting up and shutting down the reactor is in the main function of our script:

```
def poetry_main():
    addresses = parse_args()

    from twisted.internet import reactor

    poems = []

    def got_poem(poem):
        poems.append(poem)
        if len(poems) == len(addresses):
            reactor.stop()

    for address in addresses:
        host, port = address
        get_poetry(host, port, got_poem)

    reactor.run()

    for poem in poems:
        print poem
```

So if we wanted, we could take the re-usable parts and put them in a shared module that anyone could use to get their poetry (as long as they were using Twisted, of course).

By the way, when you're actually testing client 3.0 you might re-configure the poetry servers to send the poetry faster or in bigger chunks. Now that the client is less chatty in terms of output it's not as interesting to watch while it downloads the poems.

Discussion

We can visualize the callback chain at the point when a poem is delivered in Figure 11:

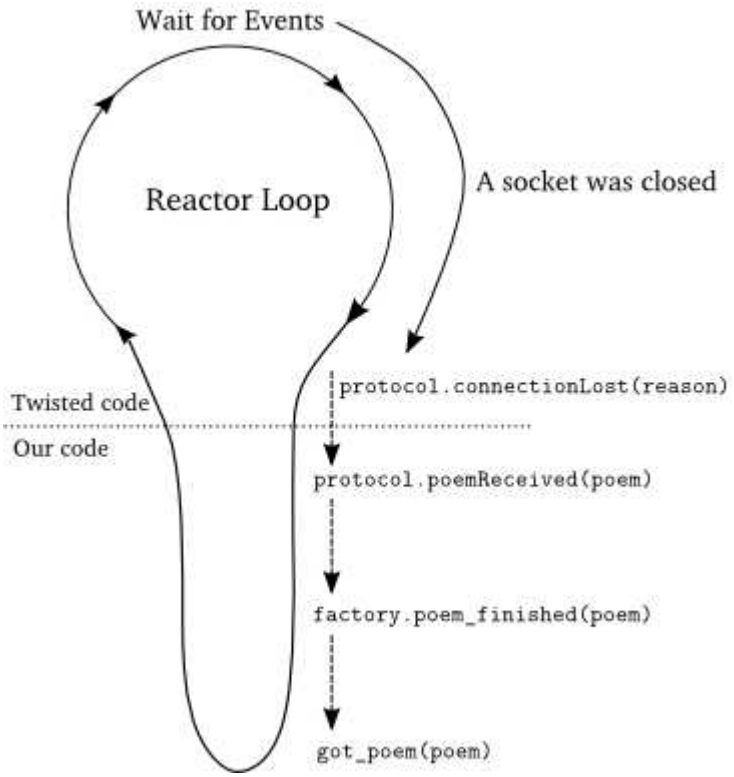

Figure 11: the poem callbacks

Figure 11 is worth contemplating. Up until now we have depicted callback chains that terminate with a single call to "our code". But when you are programming with Twisted, or any single-threaded reactive system, these callback chains might well include bits of our code making callbacks to other bits of our code. In other words, the reactive style of programming doesn't stop when it reaches code we write ourselves. In a reactor-based system, it's callbacks all the way down.

Keep that fact in mind when choosing Twisted for a project. When you make this decision:

> I'm going to use Twisted!

You are also making this decision:

> I'm going to structure my program as a series of asynchronous callback chain invocations powered by a reactor loop!

Now maybe you won't exclaim it out loud the way I do, but it is nevertheless the case. That's how Twisted works.

It's likely that most Python programs are synchronous and most Python modules are synchronous too. If we were writing a synchronous program and suddenly realized it needed some poetry, we might use the synchronous version of our `get_poetry` function by adding a few lines of code to our script like these:

```
1 ...
2 import poetrylib # I just made this module name up
3 poem = poetrylib.get_poetry(host, port)
4 ...
```

And continue on our way. If, later on, we decided we didn't really want that poem after all then we'd just snip out those lines and no one would be the wiser. But if we were writing a synchronous program and then decided to use the Twisted version of `get_poetry`, we would need to re-architect our program in the asynchronous style using callbacks. We would probably have to make significant changes to the code. Now, I'm not saying it would necessarily be a mistake to rewrite the program. It might very well make sense to do so given our requirements. But it won't be as simple as adding an `import` line and an extra

function call. Simply put, synchronous and asynchronous code do not mix.

If you are new to Twisted and asynchronous programming, I might recommend writing a few Twisted programs from scratch before you attempt to port an existing codebase. That way you will get a feel for using Twisted without the extra complexity of trying to think in both modes at once as you port from one to the other.

> If, however, your program is already asynchronous then using Twisted might be much easier. Twisted integrates relatively smoothly with pyGTK and pyQT, the Python APIs for two reactor-based GUI toolkits.

When Things Go Wrong

In client 3.0 we no longer detect a failure to connect to a poetry server, an omission which causes even more problems than it did in client 1.0. If we tell client 3.0 to download a poem from a non-existent server then instead of crashing it just waits there forever. The `clientConnectionFailed` callback still gets called, but the default implementation in the `ClientFactory` base class doesn't do anything at all. So the `got_poem` callback is never called, the reactor is never stopped, and we've got another do-nothing program like the ones we made in Part 2.

Clearly we need to handle this error, but where? The information about the failure to connect is delivered to the factory object via `clientConnectionFailed` so we'll have to start there. But this factory is supposed to be re-usable, and the proper way to handle an error will depend on the context in which the factory is being used. In some applications, missing poetry might be a disaster (No poetry?? Might as well just crash). In others, maybe we just keep on going and try to get another poem from somewhere else.

In other words, the users of `get_poetry` need to know when things go wrong, not just when they go right. In a synchronous program, `get_poetry` would raise an `Exception` and the calling code could handle it with a `try/except` statement. But in a reactive program, error conditions have to be delivered asynchronously, too. After all, we won't even find out the connection failed until after `get_poetry` returns. Here's one possibility:

```
1  def get_poetry(host, port, callback):
2      """
3      Download a poem from the given host and port and invoke
4
5        callback(poem)
6
7      when the poem is complete. If there is a failure, invoke:
8
9        callback(None)
10
11     instead.
12     """
```

By testing the callback argument (i.e., `if poem is None`) the client can determine whether we actually got a poem or not. This would suffice for our client to avoid running forever, but that approach still has some problems. First of all, using `None` to indicate failure is somewhat ad-hoc. Some asynchronous APIs might want to use `None` as a default return value instead of an error condition. Second, a `None` value carries a very limited amount of information. It can't tell us what went wrong, or include a traceback object we can use in debugging. Ok, second try:

```
1  def get_poetry(host, port, callback):
2      """
3      Download a poem from the given host and port and invoke
4
5        callback(poem)
6
7      when the poem is complete. If there is a failure, invoke:
```

```
8
9           callback(err)
10
11      instead, where err is an Exception instance.
12      """
```

Using an `Exception` is closer to what we are used to with synchronous programming. Now we can look at the exception to get more information about what went wrong and `None` is free for use as a regular value. Normally, though, when we encounter an exception in Python we also get a traceback we can analyze or print to a log for debugging at some later date. Tracebacks are extremely useful so we shouldn't give them up just because we are using asynchronous programming.

Keep in mind we don't want a traceback object for the point where our callback is invoked, that's not where the problem happened. What we really want is both the `Exception` instance and the traceback from the point where that exception was raised (assuming it was raised and not simply created).

Twisted includes an abstraction called a `Failure` that wraps up both an `Exception` and the traceback, if any, that went with it. The `Failure` docstring explains how to create one. By passing `Failure` objects to callbacks we can preserve the traceback information that's so handy for debugging.

There is some example code that uses `Failure` objects in `twisted-failure/failure-examples.py`. It shows how `Failure`s can preserve the traceback information from a raised exception, even outside the context of an `except` block. We won't dwell too much on making `Failure` instances. In Part 7 we'll see that Twisted generally ends up making them for us.

Alright, third try:

```
1   def get_poetry(host, port, callback):
2       """
3       Download a poem from the given host and port and invoke
4
5           callback(poem)
6
7       when the poem is complete. If there is a failure, invoke:
8
9           callback(err)
10
11      instead, where err is a twisted.python.failure.Failure instance.
12      """
```

With this version we get both an `Exception` and possibly a traceback record when things go wrong. Nice.

We're almost there, but we've got one more problem. Using the same callback for both normal results and failures is kind of odd. In general, we need to do quite different things on failure than on success. In a synchronous Python program we generally handle success and failure with two different code paths in a `try`/`except` statement like this:

```
1   try:
2       attempt_to_do_something_with_poetry()
3   except RhymeSchemeViolation:
4       # the code path when things go wrong
5   else:
6       # the code path when things go so, so right baby
```

If we want to preserve this style of error-handling, then we need to use a separate code path for failures. In asynchronous programming a separate code path means a separate callback:

```
1   def get_poetry(host, port, callback, errback):
2       """
3       Download a poem from the given host and port and invoke
4
```

```
        callback(poem)

    when the poem is complete. If there is a failure, invoke:

        errback(err)

    instead, where err is a twisted.python.failure.Failure instance.
    """
```

Client 3.1

Now that we have an API with reasonable error-handling semantics we can implement it. Client 3.1 is located in `twisted-client-3/get-poetry-1.py`. The changes are pretty straightforward. The `PoetryClientFactory` gets both a `callback` and an `errback`, and now it implements `clientConnectionFailed`:

```
class PoetryClientFactory(ClientFactory):

    protocol = PoetryProtocol

    def __init__(self, callback, errback):
        self.callback = callback
        self.errback = errback

    def poem_finished(self, poem):
        self.callback(poem)

    def clientConnectionFailed(self, connector, reason):
        self.errback(reason)
```

Since `clientConnectionFailed` already receives a `Failure` object (the `reason` argument) that explains why the connection failed, we just pass that along to the `errback`.

The other changes are all of a piece so I won't bother posting them here. You can test client 3.1 by using a port with no server like this:

```
python twisted-client-3/get-poetry-1.py 10004
```

And you'll get some output like this:

```
Poem failed: [Failure instance: Traceback (failure with no frames): : Connection was r
]
```

That's from the `print` statement in our `poem_failed` errback. In this case, Twisted has simply passed us an `Exception` rather than raising it, so we don't get a traceback here. But a traceback isn't really needed since this isn't a bug, it's just Twisted informing us, correctly, that we can't connect to that address.

Summary

Here's what we've learned in Part 6:

- The APIs we write for Twisted programs will have to be asynchronous.
- We can't mix synchronous code with asynchronous code.
- Thus, we have to use callbacks in our own code, just like Twisted does.
- And we have to handle errors with callbacks, too.

Does that mean every API we write with Twisted has to include two extra arguments, a callback and an errback? That doesn't sound so nice. Fortunately, Twisted has an abstraction we can use to eliminate both those arguments and pick up a few extra features in the bargain. We'll learn about it in Part 7.

Suggested Exercises

1. Update client 3.1 to timeout if the poem isn't received after a given period of time. Invoke the errback with a custom exception in that case. Don't forget to close the connection when you do.
2. Study the `trap` method on `Failure` objects. Compare it to the `except` clause in the `try/except` statement.
3. Use `print` statements to verify that `clientConnectionFailed` is called after `get_poetry` returns.

Part 7: An Interlude, Deferred

This continues the introduction started here. You can find an index to the entire series here.

Callbacks and Their Consequences

In Part 6 we came face-to-face with this fact: callbacks are a fundamental aspect of asynchronous programming with Twisted. Rather than just a way of interfacing with the reactor, callbacks will be woven into the structure of any Twisted program we write. So using Twisted, or any reactor-based asynchronous system, means organizing our code in a particular way, as a series of "callback chains" invoked by a reactor loop.

Even an API as simple as our `get_poetry` function required callbacks, two of them in fact: one for normal results and one for errors. Since, as Twisted programmers, we're going to have to make so much use of them, we should spend a little bit of time thinking about the best ways to use callbacks, and what sort of pitfalls we might encounter.

Consider this piece of code that uses the Twisted version of `get_poetry` from client 3.1:

```
...
def got_poem(poem):
    print poem
    reactor.stop()

def poem_failed(err):
    print >>sys.stderr, 'poem download failed'
    print >>sys.stderr, 'I am terribly sorry'
    print >>sys.stderr, 'try again later?'
    reactor.stop()

get_poetry(host, port, got_poem, poem_failed)

reactor.run()
```

The basic plan here is clear:

1. If we get the poem, print it out.
2. If we don't get the poem, print out an Error Haiku.
3. In either case, end the program.

The 'synchronous analogue' to the above code might look something like this:

```
...
try:
    poem = get_poetry(host, port) # the synchronous version of get_poetry
except Exception, err:
    print >>sys.stderr, 'poem download failed'
    print >>sys.stderr, 'I am terribly sorry'
    print >>sys.stderr, 'try again later?'
    sys.exit()
```

```
 9 else:
10     print poem
11     sys.exit()
```

So the callback is like the `else` block and the errback is like the `except`. That means invoking the errback is the asynchronous analogue to raising an exception and invoking the callback corresponds to the normal program flow.

What are some of the differences between the two versions? For one thing, in the synchronous version the Python interpreter will ensure that, as long as `get_poetry` raises any kind of exception at all, for any reason, the `except` block will run. If we trust the interpreter to run Python code correctly we can trust that error block to run at the right time.

Contrast that with the asynchronous version: the `poem_failed` errback is invoked by *our* code, the `clientConnectionFailed` method of the `PoetryClientFactory`. We, not Python, are in charge of making sure the error code runs if something goes wrong. So we have to make sure to handle every possible error case by invoking the errback with a `Failure` object. Otherwise, our program will become "stuck" waiting for a callback that never comes.

That shows another difference between the synchronous and asynchronous versions. If we didn't bother catching the exception in the synchronous version (by not using a `try/except`), the Python interpreter would "catch" it for us and crash to show us the error of our ways. But if we forget to "raise" our asynchronous exception (by calling the errback function in `PoetryClientFactory`), our program will just run forever, blissfully unaware that anything is amiss.

Clearly, handling errors in an asynchronous program is important, and also somewhat tricky. You might say that handling errors in asynchronous code is actually more important than handling the normal case, as things can go wrong in far more ways than they can go right. Forgetting to handle the error case is a common mistake when programming with Twisted.

Here's another fact about the synchronous code above: either the `else` block runs exactly once, or the `except` block runs exactly once (assuming the synchronous version of `get_poetry` doesn't enter an infinite loop). The Python interpreter won't suddenly decide to run them both or, on a whim, run the `else` block twenty-seven times. And it would be basically impossible to program in Python if it did!

But again, in the asynchronous case *we* are in charge of running the callback or the errback. Knowing us, we might make some mistakes. We could call both the callback and the errback, or invoke the callback twenty-seven times. That would be unfortunate for the users of `get_poetry`. Although the docstring doesn't explicitly say so, it really goes without saying that, like the `else` and `except` blocks in a `try/except` statement, either the callback will run exactly once or the errback will run exactly once, for each specific call to `get_poetry`. Either we get the poem or we don't.

Imagine trying to debug a program that makes three poetry requests and gets seven callback invocations and two errback invocations. Where would you even start? You'd probably end up writing your callbacks and errbacks to detect when they got invoked a second time for the same `get_poetry` call and throw an exception right back. Take that, `get_poetry`.

One more observation: both versions have some duplicate code. The asynchronous version has two calls to `reactor.stop` and the synchronous version has two calls to `sys.exit`. We might refactor the synchronous version like this:

```
1 ...
2 try:
3     poem = get_poetry(host, port) # the synchronous version of get_poetry
4 except Exception, err:
5     print >>sys.stderr, 'poem download failed'
6     print >>sys.stderr, 'I am terribly sorry'
7     print >>sys.stderr, 'try again later?'
```

```
 8 else:
 9     print poem
10
11 sys.exit()
```

Can we refactor the asynchronous version in a similar way? It's not really clear that we can, since the callback and errback are two different functions. Do we have to go back to a single callback to make this possible?

Ok, here are some of the insights we've discovered about programming with callbacks:

1. Calling errbacks is very important. Since errbacks take the place of `except` blocks, users need to be able to count on them. They aren't an optional feature of our APIs.
2. *Not* invoking callbacks at the wrong time is just as important as calling them at the right time. For a typical use case, the callback and errback are mutually exclusive and invoked exactly once.
3. Refactoring common code might be harder when using callbacks.

We'll have more to say about callbacks in future Parts, but for now this is enough to see why Twisted might have an abstraction devoted to managing them.

The Deferred

Since callbacks are used so much in asynchronous programming, and since using them correctly can, as we have discovered, be a bit tricky, the Twisted developers created an abstraction called a `Deferred` to make programming with callbacks easier. The `Deferred` class is defined in `twisted.internet.defer`.

> The word "deferred" is either a verb or an adjective in everyday English, so it might sound a little strange used as a noun. Just know that, from now on, when I use the phrase "the deferred" or "a deferred", I'm referring to an instance of the `Deferred` class. We'll talk about why it is called `Deferred` in a future Part. It might help to mentally add the word "result" to each phrase, as in "the deferred result". As we will eventually see, that's really what it is.

A deferred contains a pair of callback chains, one for normal results and one for errors. A newly-created deferred has two empty chains. We can populate the chains by adding callbacks and errbacks and then *fire* the deferred with either a normal result (here's your poem!) or an exception (I couldn't get the poem, and here's why). Firing the deferred will invoke the appropriate callbacks or errbacks in the order they were added. Figure 12 illustrates a deferred instance with its callback/errback chains:

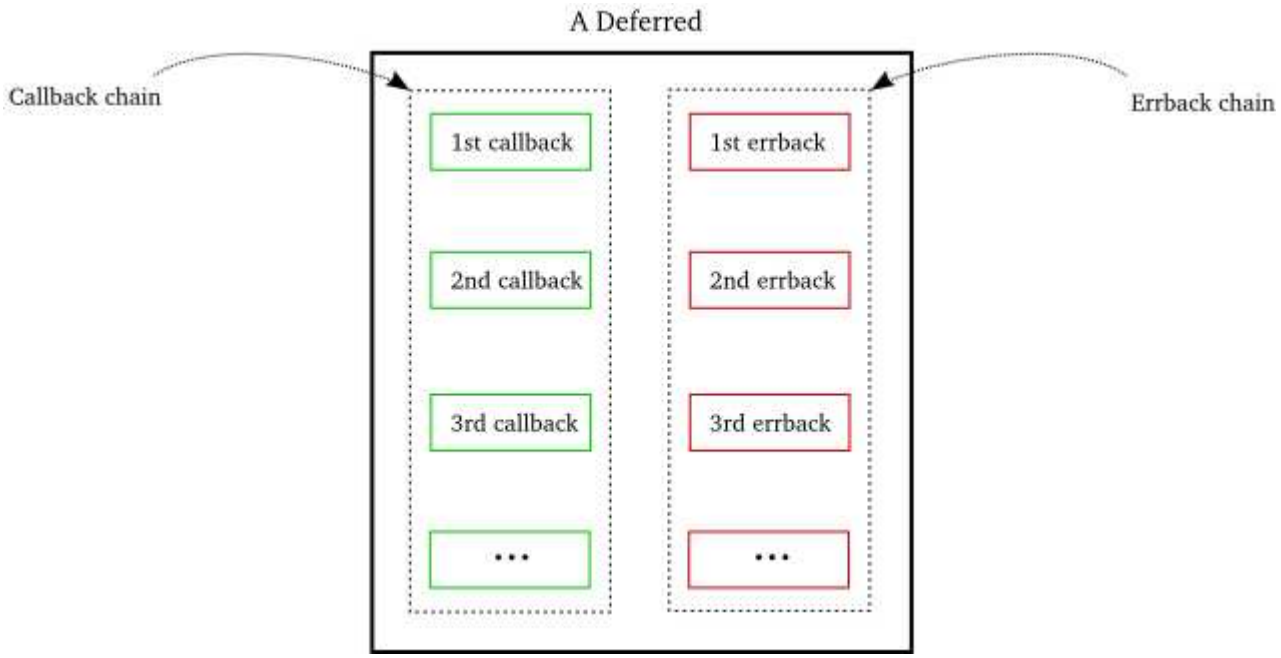

Figure 12: A Deferred

Let's try this out. Since deferreds don't use the reactor, we can test them out without starting up the loop.

> You might have noticed a method on `Deferred` called `setTimeout` that does use the reactor. It is deprecated and will cease to exist in a future release. Pretend it's not there and don't use it.

Our first example is in twisted-deferred/defer-1.py:

```
1  from twisted.internet.defer import Deferred
2
3  def got_poem(res):
4      print 'Your poem is served:'
5      print res
6
7  def poem_failed(err):
8      print 'No poetry for you.'
9
10 d = Deferred()
11
12 # add a callback/errback pair to the chain
13 d.addCallbacks(got_poem, poem_failed)
14
15 # fire the chain with a normal result
16 d.callback('This poem is short.')
17
18 print "Finished"
```

This code makes a new deferred, adds a callback/errback pair with the `addCallbacks` method, and then fires the "normal result" chain with the `callback` method. Of course, it's not much of a chain since it only has a single callback, but no matter. Run the code and it produces this output:

```
Your poem is served:
This poem is short.
Finished
```

That's pretty simple. Here are some things to notice:

1. Just like the callback/errback pairs we used in client 3.1, the callbacks we add to this deferred each take one argument, either a normal result or an error result. It turns out that deferreds support callbacks and errbacks with multiple arguments, but they always have at least one, and the first argument is always either a normal result or an error result.
2. We add callbacks and errbacks to the deferred in pairs.
3. The `callback` method fires the deferred with a normal result, the method's only argument.
4. Looking at the order of the `print` output, we can see that firing the deferred invokes the callbacks immediately. There's nothing asynchronous going on at all. There can't be, since no reactor is running. It really boils down to an ordinary Python function call.

Ok, let's push the other button. The example in twisted-deferred/defer-2.py fires the deferred's errback chain:

```
1  from twisted.internet.defer import Deferred
2  from twisted.python.failure import Failure
3
4  def got_poem(res):
5      print 'Your poem is served:'
6      print res
7
8  def poem_failed(err):
9      print 'No poetry for you.'
10
11 d = Deferred()
```

```
12
13  # add a callback/errback pair to the chain
14  d.addCallbacks(got_poem, poem_failed)
15
16  # fire the chain with an error result
17  d.errback(Failure(Exception('I have failed.')))
18
19  print "Finished"
```

And after running that script we get this output:

```
No poetry for you.
Finished
```

So firing the errback chain is just a matter of calling the `errback` method instead of the `callback` method, and the method argument is the error result. And just as with callbacks, the errbacks are invoked immediately upon firing.

In the previous example we are passing a `Failure` object to the `errback` method like we did in client 3.1. That's just fine, but a deferred will turn ordinary `Exception`s into `Failure`s for us. We can see that with twisted-deferred/defer-3.py:

```
1  from twisted.internet.defer import Deferred
2
3  def got_poem(res):
4      print 'Your poem is served:'
5      print res
6
7  def poem_failed(err):
8      print err.__class__
9      print err
10     print 'No poetry for you.'
11
12 d = Deferred()
13
14 # add a callback/errback pair to the chain
15 d.addCallbacks(got_poem, poem_failed)
16
17 # fire the chain with an error result
18 d.errback(Exception('I have failed.'))
```

Here we are passing a regular `Exception` to the `errback` method. In the errback, we are printing out the class and the error result itself. We get this output:

```
twisted.python.failure.Failure
[Failure instance: Traceback (failure with no frames): : I have failed.
]
No poetry for you.
```

This means when we use deferreds we can go back to working with ordinary `Exception`s and the `Failure`s will get created for us automatically. A deferred will guarantee that each errback is invoked with an actual `Failure` instance.

We tried pressing the `callback` button and we tried pressing the `errback` button. Like any good engineer, you probably want to start pressing them over and over. To make the code shorter, we'll use the same function for both the callback and the errback. Just remember they get different return values; one is a result and the other is a failure. Check out twisted-deferred/defer-4.py:

```
1  from twisted.internet.defer import Deferred
2  def out(s): print s
3  d = Deferred()
4  d.addCallbacks(out, out)
5  d.callback('First result')
```

```
6 | d.callback('Second result')
7 | print 'Finished'
```

Now we get this output:

```
First result
Traceback (most recent call last):
  ...
twisted.internet.defer.AlreadyCalledError
```

This is interesting! A deferred will not let us fire the normal result callbacks a second time. In fact, a deferred cannot be fired a second time no matter what, as demonstrated by these examples:

- twisted-deferred/defer-4.py
- twisted-deferred/defer-5.py
- twisted-deferred/defer-6.py
- twisted-deferred/defer-7.py

Notice those final `print` statements are never called. The `callback` and `errback` methods are raising genuine `Exception`s to let us know we've already fired that deferred. Deferreds help us avoid one of the pitfalls we identified with callback programming. When we use a deferred to manage our callbacks, we simply can't make the mistake of calling both the callback and the errback, or invoking the callback twenty-seven times. We can try, but the deferred will raise an exception right back at us, instead of passing our mistake onto the callbacks themselves.

Can deferreds help us to refactor asynchronous code? Consider the example in twisted-deferred/defer-8.py:

```
1  | import sys
2  |
3  | from twisted.internet.defer import Deferred
4  |
5  | def got_poem(poem):
6  |     print poem
7  |     from twisted.internet import reactor
8  |     reactor.stop()
9  |
10 | def poem_failed(err):
11 |     print >>sys.stderr, 'poem download failed'
12 |     print >>sys.stderr, 'I am terribly sorry'
13 |     print >>sys.stderr, 'try again later?'
14 |     from twisted.internet import reactor
15 |     reactor.stop()
16 |
17 | d = Deferred()
18 |
19 | d.addCallbacks(got_poem, poem_failed)
20 |
21 | from twisted.internet import reactor
22 |
23 | reactor.callWhenRunning(d.callback, 'Another short poem.')
24 |
25 | reactor.run()
```

This is basically our original example above, with a little extra code to get the reactor going. Notice we are using `callWhenRunning` to fire the deferred after the reactor starts up. We're taking advantage of the fact that `callWhenRunning` accepts additional positional- and keyword-arguments to pass to the callback when it is run. Many Twisted APIs that register callbacks follow this same convention, including the APIs to add callbacks to deferreds.

Both the callback and the errback stop the reactor. Since deferreds support chains of callbacks and

errbacks, we can refactor the common code into a second link in the chains, a technique illustrated in twisted-deferred/defer-9.py:

```
import sys

from twisted.internet.defer import Deferred

def got_poem(poem):
    print poem

def poem_failed(err):
    print >>sys.stderr, 'poem download failed'
    print >>sys.stderr, 'I am terribly sorry'
    print >>sys.stderr, 'try again later?'

def poem_done(_):
    from twisted.internet import reactor
    reactor.stop()

d = Deferred()

d.addCallbacks(got_poem, poem_failed)
d.addBoth(poem_done)

from twisted.internet import reactor

reactor.callWhenRunning(d.callback, 'Another short poem.')

reactor.run()
```

The `addBoth` method adds the same function to both the callback and errback chains. And we can refactor asynchronous code after all.

> **Note:** there is a subtlety in the way this deferred would actually execute its errback chain. We'll discuss it in a future Part, but keep in mind there is more to learn about deferreds.

Summary

In this Part we analyzed callback programming and identified some potential problems. We also saw how the `Deferred` class can help us out:

1. We can't ignore errbacks, they are required for any asynchronous API. Deferreds have support for errbacks built in.
2. Invoking callbacks multiple times will likely result in subtle, hard-to-debug problems. Deferreds can only be fired once, making them similar to the familiar semantics of `try/except` statements.
3. Programming with plain callbacks can make refactoring tricky. With deferreds, we can refactor by adding links to the chain and moving code from one link to another.

We're not done with the story of deferreds, there are more details of their rationale and behavior to explore. But we've got enough to start using them in our poetry client, so we'll do that in Part 8.

Suggested Exercises

1. The last example ignores the argument to `poem_done`. Print it out instead. Make `got_poem` return a value and see how that changes the argument to `poem_done`.
2. Modify the last two deferred examples to fire the errback chains. Make sure to fire the errback with an `Exception`.
3. Read the docstrings for the `addCallback` and `addErrback` methods on `Deferred`.

Part 8: Deferred Poetry

This continues the introduction started here. You can find an index to the entire series here.

Client 4.0

Now that we have know something about deferreds, we can rewrite our Twisted poetry client to use them. You can find client 4.0 in `twisted-client-4/get-poetry.py`.

Our `get_poetry` function no longer needs `callback` or `errback` arguments. Instead, it returns a new deferred to which the user may attach callbacks and errbacks as needed.

```
def get_poetry(host, port):
    """
    Download a poem from the given host and port. This function
    returns a Deferred which will be fired with the complete text of
    the poem or a Failure if the poem could not be downloaded.
    """
    d = defer.Deferred()
    from twisted.internet import reactor
    factory = PoetryClientFactory(d)
    reactor.connectTCP(host, port, factory)
    return d
```

Our factory object is initialized with a deferred instead of a callback/errback pair. Once we have the poem, or we find out we couldn't connect to the server, the deferred is fired with either a poem or a failure:

```
class PoetryClientFactory(ClientFactory):

    protocol = PoetryProtocol

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

    def poem_finished(self, poem):
        if self.deferred is not None:
            d, self.deferred = self.deferred, None
            d.callback(poem)

    def clientConnectionFailed(self, connector, reason):
        if self.deferred is not None:
            d, self.deferred = self.deferred, None
            d.errback(reason)
```

Notice the way we release our reference to the deferred after it is fired. This is a pattern found in several places in the Twisted source code and helps to ensure we do not fire the same deferred twice. It makes life a little easier for the Python garbage collector, too.

Once again, there is no need to change the `PoetryProtocol`, it's just fine as is. All that remains is to update the `poetry_main` function:

```
def poetry_main():
    addresses = parse_args()

    from twisted.internet import reactor

    poems = []
    errors = []

    def got_poem(poem):
```

```
10          poems.append(poem)
11
12      def poem_failed(err):
13          print >>sys.stderr, 'Poem failed:', err
14          errors.append(err)
15
16      def poem_done(_):
17          if len(poems) + len(errors) == len(addresses):
18              reactor.stop()
19
20      for address in addresses:
21          host, port = address
22          d = get_poetry(host, port)
23          d.addCallbacks(got_poem, poem_failed)
24          d.addBoth(poem_done)
25
26      reactor.run()
27
28      for poem in poems:
29          print poem
```

Notice how we take advantage of the chaining capabilities of the deferred to refactor the `poem_done` invocation out of our primary callback and errback.

Because deferreds are used so much in Twisted code, it's common practice to use the single-letter local variable `d` to hold the deferred you are currently working on. For longer term storage, like object attributes, the name "`deferred`" is often used.

Discussion

With our new client the asynchronous version of `get_poetry` accepts the same information as our synchronous version, just the address of the poetry server. The synchronous version returns a poem, while the asynchronous version returns a deferred. Returning a deferred is typical of the asynchronous APIs in Twisted and programs written with Twisted, and this points to another way of conceptualizing deferreds:

> A `Deferred` object represents an "asynchronous result" or a "result that has not yet come".

We can contrast these two styles of programming in Figure 13:

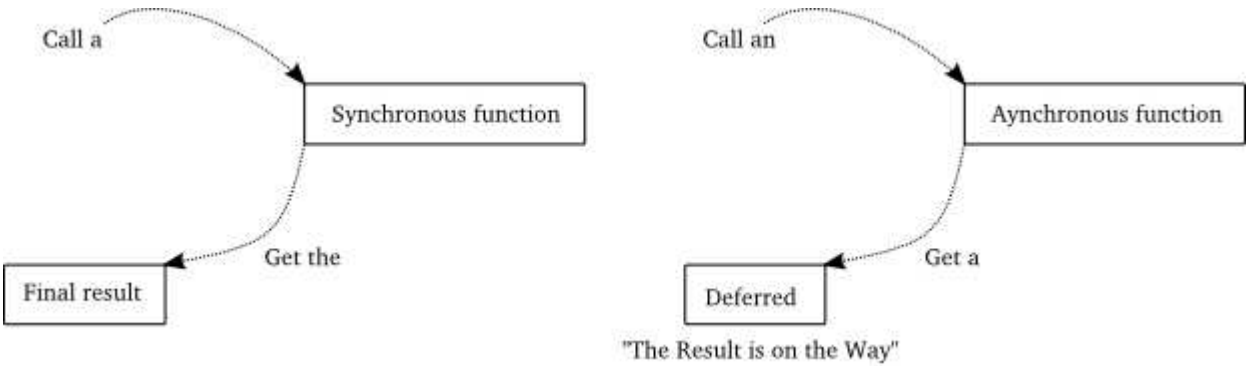

Figure 13: sync versus async

By returning a deferred, an asynchronous API is giving this message to the user:

> I'm an asynchronous function. Whatever you want me to do might not be done yet. But when it is done, I'll fire the callback chain of this deferred with the result. On the other hand, if something goes wrong, I'll fire the errback chain of this deferred instead.

Of course, that function itself won't literally fire the deferred, it has already returned. Rather, the function

has set in motion a chain of events that will eventually result in the deferred being fired.

So deferreds are a way of "time-shifting" the results of functions to accommodate the needs of the asynchronous model. And a deferred returned by a function is a notice that the function is asynchronous, the embodiment of the future result, and a promise that the result will be delivered.

> It is possible for a synchronous function to return a deferred, so technically a deferred return value means the function is potentially asynchronous. We'll see examples of synchronous functions returning deferreds in future Parts.

Because the behavior of deferreds is well-defined and well-known (to folks with some experience programming with Twisted), by returning deferreds from your own APIs you are making it easier for other Twisted programmers to understand and use your code. Without deferreds, each Twisted program, or even each internal Twisted component, might have its own unique method for managing callbacks that you would have to learn in order to use it.

When You're Using Deferreds, You're Still Using Callbacks, and They're Still Invoked by the Reactor

When first learning Twisted, it is a common mistake to attribute more functionality to deferreds than they actually have. Specifically, it is often assumed that adding a function to a deferred's chain automatically makes that function asynchronous. This might lead you to think you could use, say, `os.system` with Twisted by adding it to a deferred with `addCallback`.

I think this mistake is caused by trying to learn Twisted without first learning the asynchronous model. Since typical Twisted code uses lots of deferreds and only occasionally refers to the reactor, it can appear that deferreds are doing all the work. If you have read this introduction from the beginning, it should be clear this is far from the case. Although Twisted is composed of many parts that work together, the primary responsibility for implementing the asynchronous model falls to the reactor. Deferreds are a useful abstraction, but we wrote several versions of our Twisted client without using them in any way.

Let's look at a stack trace at the point when our first callback is invoked. Run the example program in twisted-client-4/get-poetry-stack.py with the address of a running poetry server. You should get some output like this:

```
  File "twisted-client-4/get-poetry-stack.py", line 129, in
    poetry_main()
  File "twisted-client-4/get-poetry-stack.py", line 122, in poetry_main
    reactor.run()

  ... # some more Twisted function calls

    protocol.connectionLost(reason)
  File "twisted-client-4/get-poetry-stack.py", line 59, in connectionLost
    self.poemReceived(self.poem)
  File "twisted-client-4/get-poetry-stack.py", line 62, in poemReceived
    self.factory.poem_finished(poem)
  File "twisted-client-4/get-poetry-stack.py", line 75, in poem_finished
    d.callback(poem) # here's where we fire the deferred

  ... # some more methods on Deferreds

  File "twisted-client-4/get-poetry-stack.py", line 105, in got_poem
    traceback.print_stack()
```

That's pretty similar to the stack trace we created for client 2.0. We can visualize the latest trace in Figure 14:

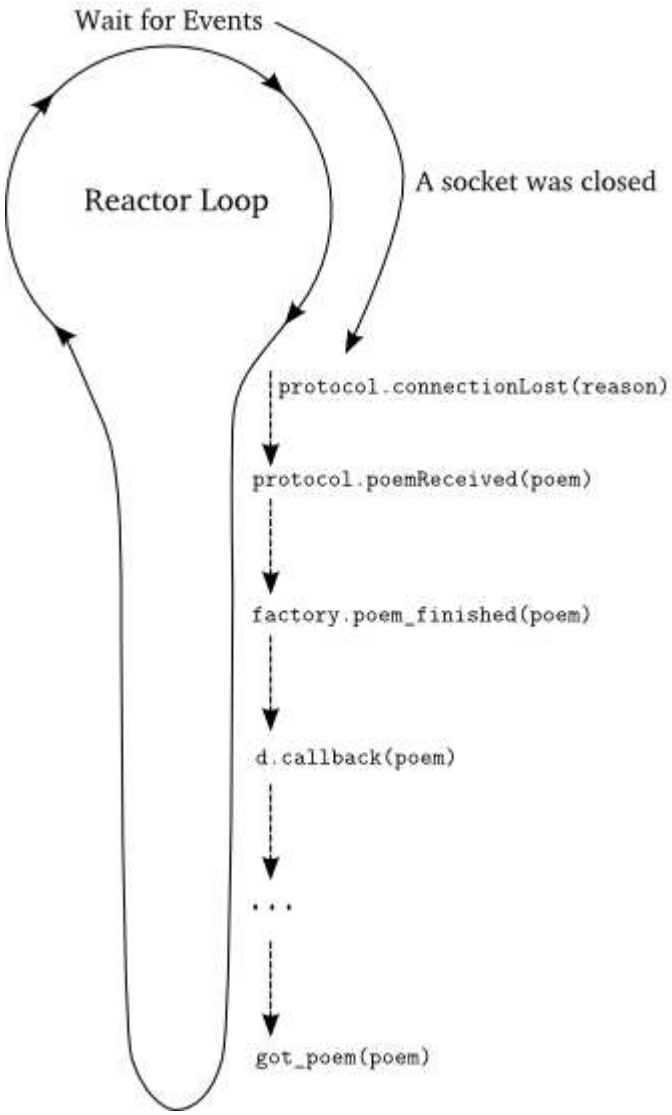

Figure 14: A callback with a deferred

Again, this is similar to our previous Twisted clients, though the visual representation is starting to become vaguely disturbing. We probably won't be showing any more of these, for the sake of the children. One wrinkle not reflected in the figure: the callback chain above doesn't return control to the reactor until the second callback in the deferred (`poem_done`) is invoked, which happens right after the first callback (`got_poem`) returns.

> There's one more difference with our new stack trace. The line separating "Twisted code" from "our code" is a little fuzzier, since the methods on deferreds are really Twisted code. This interleaving of Twisted and user code in a callback chain is common in larger Twisted programs which make extensive use of other Twisted abstractions.

By using a deferred we've added a few more steps in the callback chain that starts in the Twisted reactor, but we haven't changed the fundamental mechanics of the asynchronous model. Recall these facts about callback programming:

1. Only one callback runs at a time.
2. When the reactor is running our callbacks are not.
3. And vice-versa.
4. If our callback blocks then the whole program blocks.

Attaching a callback to a deferred doesn't change these facts in any way. In particular, a callback that blocks will still block if it's attached to a deferred. So that deferred will block when it is fired

(`d.callback`), and thus Twisted will block. And we conclude:

> Deferreds are a solution (a particular one invented by the Twisted developers) to the problem of managing callbacks. They are neither a way of avoiding callbacks nor a way to turn blocking callbacks into non-blocking callbacks.

We can confirm the last point by constructing a deferred with a blocking callback. Consider the example code in `twisted-deferred/defer-block.py`. The second callback blocks using the `time.sleep` function. If you run that script and examine the order of the `print` statements, it will be clear that a blocking callback also blocks inside a deferred.

Summary

By returning a `Deferred`, a function tells the user "I'm asynchronous" and provides a mechanism (add your callbacks and errbacks here!) to obtain the asynchronous result when it arrives. Deferreds are used extensively throughout the Twisted codebase and as you explore Twisted's APIs you are bound to keep encountering them. So it will pay to become familiar with deferreds and comfortable in their use.

Client 4.0 is the first version of our Twisted poetry client that's truly written in the "Twisted style", using a deferred as the return value of an asynchronous function call. There are a few more Twisted APIs we could use to make it a little cleaner, but I think it represents a pretty good example of how simple Twisted programs are written, at least on the client side. Eventually we'll re-write our poetry server using Twisted, too.

But we're not quite finished with deferreds. For a relatively short piece of code, the `Deferred` class provides a surprising number of features. We'll talk about some more of those features, and their motivation, in Part 9.

Suggested Exercises

1. Update client 4.0 to timeout if the poem isn't received after a given period of time. Fire the deferred's errback with a custom exception in that case. Don't forget to close the connection when you do.
2. Update client 4.0 to print out the appropriate server address when a poem download fails, so the user can tell which server is the culprit. Don't forget you can add extra positional- and keyword-arguments when you attach callbacks and errbacks.

Part 9: A Second Interlude, Deferred

This continues the introduction started here. You can find an index to the entire series here.

More Consequence of Callbacks

We're going to pause for a moment to think about callbacks again. Although we now know enough about deferreds to write simple asynchronous programs in the Twisted style, the `Deferred` class provides more features that only come into play in more complex settings. So we're going to think up some more complex settings and see what sort of challenges they pose when programming with callbacks. Then we'll investigate how deferreds address those challenges.

To motivate our discussion we're going to add a hypothetical feature to our poetry client. Suppose some hard-working Computer Science professor has invented a new poetry-related algorithm, the Byronification Engine. This nifty algorithm takes a single poem as input and produces a new poem like the original, but written in the style of Lord Byron. What's more, our professor has kindly provided a reference implementation in Python, with this interface:

```
class IByronificationEngine(Interface):

    def byronificate(poem):
        """
        Return a new poem like the original, but in the style of Lord Byron.

        Raises GibberishError if the input is not a genuine poem.
        """
```

Like most bleeding-edge software, the implementation has some bugs. This means that in addition to the documented exception, the `byronificate` method sometimes throws random exceptions when it hits a corner-case the professor forgot to handle.

We'll also assume the engine runs fast enough that we can just call it in the main thread without worrying about tying up the reactor. This is how we want our program to work:

1. Try to download the poem.
2. If the download fails, tell the user we couldn't get the poem.
3. If we do get the poem, transform it with the Byronification Engine.
4. If the engine throws a `GibberishError`, tell the user we couldn't get the poem.
5. If the engine throws another exception, just keep the original poem.
6. If we have a poem, print it out.
7. End the program.

The idea here is that a `GibberishError` means we didn't get an actual poem after all, so we'll just tell the user the download failed. That's not so useful for debugging, but our users just want to know whether we got a poem or not. On the other hand, if the engine fails for some other reason then we'll use the poem we got from the server. After all, some poetry is better than none at all, even if it's not in the trademark Byron style.

Here's the synchronous version of our code:

```
try:
    poem = get_poetry(host, port) # synchronous get_poetry
except:
    print >>sys.stderr, 'The poem download failed.'
else:
    try:
        poem = engine.byronificate(poem)
    except GibberishError:
        print >>sys.stderr, 'The poem download failed.'
    except:
        print poem # handle other exceptions by using the original poem
    else:
        print poem

sys.exit()
```

This sketch of a program could be make simpler with some refactoring, but it illustrates the flow of logic pretty clearly. We want to update our most recent poetry client (which uses deferreds) to implement this same scheme. But we won't do that until Part 10. For now, instead, let's imagine how we might do this with client 3.1, our last client that didn't use deferreds at all. Suppose we didn't bother handling exceptions, but instead just changed the `got_poem` callback like this:

```
def got_poem(poem):
    poems.append(byron_engine.byronificate(poem))
    poem_done()
```

What happens when the `byronificate` method raises a `GibberishError` or some other exception? Looking at Figure 11 from Part 6, we can see that:

1. The exception will propagate to the `poem_finished` callback in the factory, the method that actually invokes the callback.
2. Since `poem_finished` doesn't catch the exception, it will proceed to `poemReceived` on the protocol.
3. And then on to `connectionLost`, also on the protocol.
4. And then up into the core of Twisted itself, finally ending up at the reactor.

As we have learned, the reactor will catch and log the exception instead of crashing. But what it certainly won't do is tell the user we couldn't download a poem. The reactor doesn't know anything about poems or `GibberishError`s, it's a general-purpose piece of code used for all kinds of networking, even non-poetry-related networking.

Notice how, at each step in the list above, the exception moves to a more general-purpose piece of code than the one before. And at no step after `got_poem` is the exception in a piece of code that could be expected to handle an error in the specific way we want for this client. This situation is basically the exact opposite of the way exceptions propagate in synchronous code.

Take a look at Figure 15, an illustration of a call stack we might see with a synchronous poetry client :

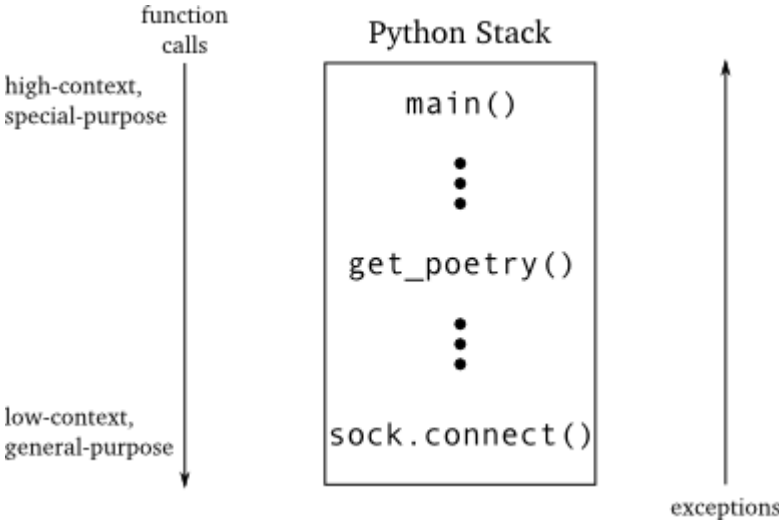

Figure 15: synchronous code and exceptions

The `main` function is "high-context", meaning it knows a lot about the whole program, why it exists, and how it's supposed to behave overall. Typically, `main` would have access to the command-line options that indicate just how the user wants the program to work (and perhaps what to do if something goes wrong). It also has a very specific purpose: running the show for a command-line poetry client.

The socket `connect` method, on the other hand, is "low-context". All it knows is that it's supposed to connect to some network address. It doesn't know what's on the other end or why we need to connect right now. But `connect` is quite general-purpose — you can use it no matter what sort of service you are connecting to.

And `get_poetry` is in the middle. It knows it's getting some poetry (and that's the only thing it's really good at), but not what should happen if it can't.

So an exception thrown by `connect` will move up the stack, from low-context and general-purpose code to high-context and special-purpose code, until it reaches some code with enough context to know what to do when something goes wrong (or it hits the Python interpreter and the program crashes).

Of course the exception is really just moving up the stack no matter what rather than literally seeking out high-context code. It's just that in a typical synchronous program "up the stack" and "towards higher-context" are the same direction.

Now recall our hypothetical modification to client 3.1 above. The call stack we analyzed is pictured in Figure 16, abbreviated to just a few functions:

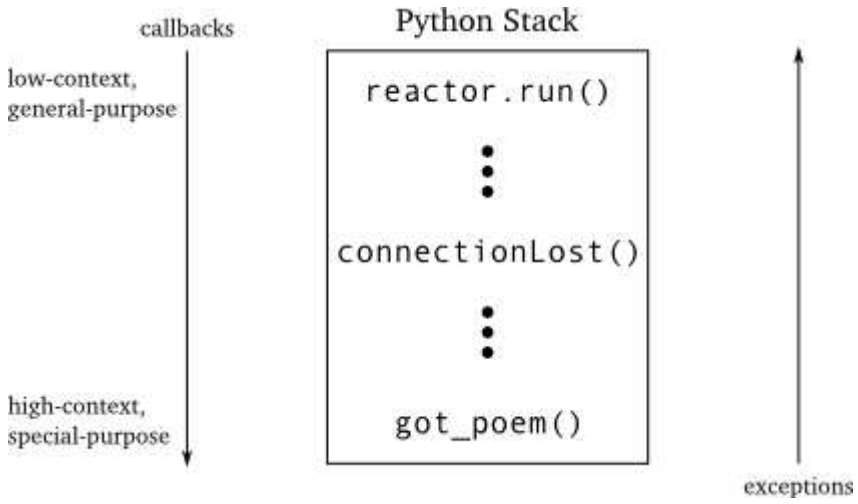

Figure 16: asynchronous callbacks and exceptions

The problem is now clear: during a callback, low-context code (the reactor) is calling higher-context code which may in turn call even higher-context code, and so on. So if an exception occurs and it isn't handled immediately, close to the same stack frame where it occurred, it's unlikely to be handled at all. Because each time the exception moves up the stack it moves to a piece of lower-context code that's even less likely to know what to do.

Once an exception crosses over into the Twisted core the game is up. The exception will not be handled, it will only be noted (when the reactor finally catches it). So when we are programming with "plain old" callbacks (without using deferreds), we must be careful to catch every exception before it gets back into Twisted proper, at least if we want to have any chance of handling errors according to our own rules. And that includes exceptions caused by our own bugs!

Since a bug can exist anywhere in our code, we would need to wrap every callback we write in an extra "outer layer" of `try/except` statements so the exceptions from our fumble-fingered typos can be handled as well. And the same goes for our errbacks because code to handle errors can have bugs too.

Well that's not so nice.

The Fine Structure of Deferreds

It turns out the `Deferred` class helps us solve this problem. Whenever a deferred invokes a callback or errback, it catches any exception that might be raised. In other words, a deferred acts as the "outer layer" of `try/except` statements so we don't need to write that layer after all, as long as we use deferreds. But what does a deferred do with an exception it catches? Simple — it passes the exception (in the form of a `Failure`) to the next errback in the chain.

So the first errback we add to a deferred is there to handle whatever error condition is signaled when the deferred's `.errback(err)` method is called. But the second errback will handle any exception raised by either the first callback or the first errback, and so on down the line.

Recall Figure 12, a visual representation of a deferred with some callbacks and errbacks in the chain. Let's call the first callback/errback pair stage 0, the next pair stage 1, and so on.

At a given stage **N**, if either the callback or the errback (whichever was executed) fails, then the errback in stage **N+1** is called with the appropriate `Failure` object and the callback in stage **N+1** is *not* called.

By passing exceptions raised by callbacks "down the chain", a deferred moves exceptions in the direction of "higher context". This also means that invoking the `callback` and `errback` methods of a deferred will never result in an exception for the caller (as long as you only fire the deferred once!), so lower-level code can safely fire a deferred without worrying about catching exceptions. Instead, higher-level code catches the exception by adding errbacks to the deferred (with `addErrback`, etc.).

Now in synchronous code, an exception stops propagating as soon as it is caught. So how does an errback signal the fact that it "caught" the error? Also simple — by *not* raising an exception. And in that case, the execution switches over to the callback line. So at a given stage **N**, if either the callback or errback succeeds (i.e., doesn't raise an exception) then the callback in stage **N+1** is called with the return value from stage **N**, and the errback in stage **N+1** is *not* called.

Let's summarize what we know about the deferred firing pattern:

1. A deferred contains a chain of ordered callback/errback pairs (stages). The pairs are in the order they were added to the deferred.
2. Stage 0, the first callback/errback pair, is invoked when the deferred is fired. If the deferred is fired with the `callback` method, then the stage 0 callback is called. If the deferred is fired with the `errback` method, then the stage 0 errback is called.
3. If stage **N** fails, then the stage **N+1** errback is called with the exception (wrapped in a `Failure`) as the first argument.
4. If stage **N** succeeds, then the stage **N+1** callback is called with the stage **N** return value as the first argument.

This pattern is illustrated in Figure 17:

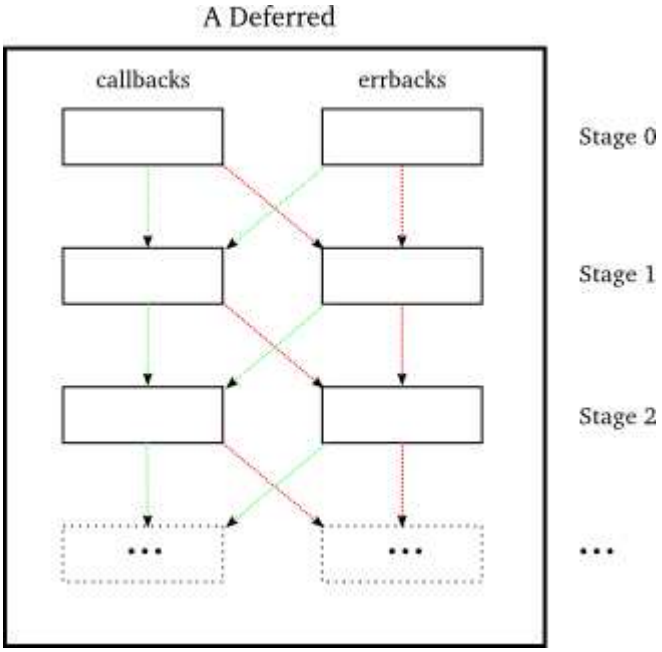

Figure 17: control flow in a deferred

The green lines indicate what happens when a callback or errback succeeds and the red lines are for failures. The lines show both the flow of control and the flow of exceptions and return values down the chain. Figure 17 shows all possible paths a deferred might take, but only one path will be taken in any particular case. Figure 18 shows one possible path for a "firing":

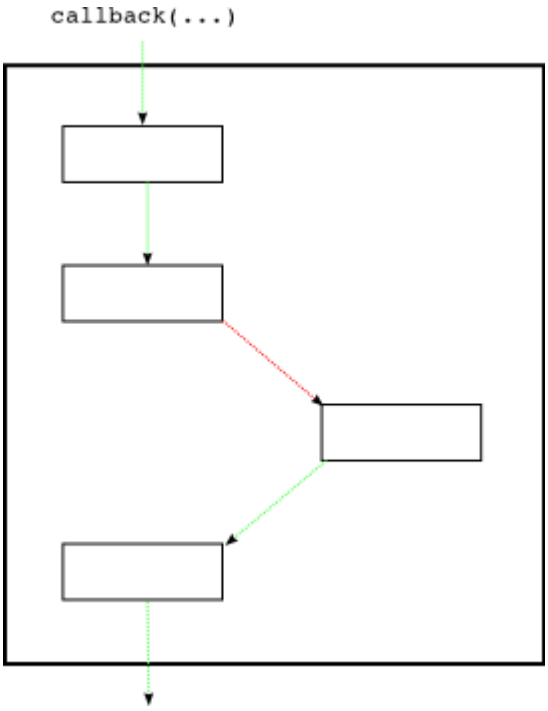

Figure 18: one possible deferred firing pattern

In figure 18, the deferred's `callback` method is called, which invokes the callback in stage 0. That callback succeeds, so control (and the return value from stage 0) passes to the stage 1 callback. But that callback fails (raises an exception), so control switches to the errback in stage 2. The errback "handles" the error (it doesn't raise an exception) so control moves back to the callback chain and the callback in stage 3 is called with the result from the stage 2 errback.

Notice that any path you can make with Figure 17 will pass through every stage in the chain, but only one member of the callback/errback pair at any stage will be called.

In Figure 18, we've indicated that the stage 3 callback succeeds by drawing a green arrow out of it, but since there aren't any more stages in that deferred, the result of stage 3 doesn't really go anywhere. If the callback succeeds, that's not really a problem, but what if it had failed? If the last stage in a deferred fails, then we say the failure is *unhandled*, since there is no errback to "catch" it.

In synchronous code an unhandled exception will crash the interpreter, and in plain-old-callbacks asynchronous code an unhandled exception is caught by the reactor and logged. What happens to unhandled exceptions in deferreds? Let's try it out and see. Look at the sample code in [twisted-deferred/defer-unhandled.py](). That code is firing a deferred with a single callback that always raises an exception. Here's the output of the program:

```
Finished
Unhandled error in Deferred:
Traceback (most recent call last):
  ...
--- <exception caught here> ---
  ...
exceptions.Exception: oops
```

Some things to notice:

1. The last `print` statement runs, so the program is not "crashed" by the exception.
2. That means the Traceback is just getting printed out, it's not crashing the interpreter.
3. The text of the traceback tells us where the deferred itself caught the exception.
4. The "Unhandled" message gets printed out after "Finished".

So when you use deferreds, unhandled exceptions in callbacks will still be noted, for debugging purposes, but as usual they won't crash the program (in fact they won't even make it to the reactor, the deferred will catch them first). By the way, the reason that "Finished" comes first is because the "Unhandled" message isn't actually printed until the deferred is garbage collected. We'll see the reason for that in a future Part.

Now, in synchronous code we can "re-raise" an exception using the `raise` keyword without any arguments. Doing so raises the original exception we were handling and allows us to take some action on an error without completely handling it. It turns out we can do the same thing in an errback. A deferred will consider a callback/errback to have failed if:

- The callback/errback raises any kind of exception, or
- The callback/errback returns a `Failure` object.

Since an errback's first argument is always a `Failure`, an errback can "re-raise" the exception by returning its first argument, after performing whatever action it wants to take.

Callbacks and Errbacks, Two by Two

One thing that should be clear from the above discussion is that the order you add callbacks and errbacks to a deferred makes a big difference in how the deferred will fire. What should also be clear is that, in a deferred, callbacks and errbacks always occur in pairs. There are four methods on the `Deferred` class you can use to add pairs to the chain:

1. `addCallbacks`
2. `addCallback`
3. `addErrback`
4. `addBoth`

Obviously, the first and last methods add a pair to the chain. But the middle two methods also add a callback/errback pair. The `addCallback` method adds an explicit callback (the one you pass to the method) and an implicit "pass-through" errback. A pass-through function is a dummy function that just returns its first argument. Since the first argument to an errback is always a `Failure`, a pass-through errback will always "fail" and send its error to the next errback in the chain.

As you've no doubt guessed, the `addErrback` function adds an explicit errback and an implicit pass-through callback. And since the first argument to a callback is never a `Failure`, a pass-through callback sends its result to the next callback in the chain.

The Deferred Simulator

It's a good idea to become familiar with the way deferreds fire their callbacks and errbacks. The python script in `twisted-deferred/deferred-simulator.py` is a "deferred simulator", a little python program that lets you explore how deferreds fire. When you run the script it will ask you to enter list of callback and errback pairs, one per line. For each callback or errback, you specify that either:

- It returns a given value (succeds), or
- It raises a given exception (fails), or
- It returns its argument (passthru).

After you've entered all the pairs you want to simulate, the script will print out, in high-resolution ASCII art, a diagram showing the contents of the chain and the firing patterns for the `callback` and `errback` methods. You will want to use a terminal window that is as wide as possible to see everything correctly. You can also use the `--narrow` option to print the diagrams one after the other, but it's easier to see their relationships when you print them side-by-side.

Of course, in real code a callback isn't going to return the same value every time, and a given function might sometimes succeed and other times fail. But the simulator can give you a picture of what will happen for a given combination of normal results and failures, in a given arrangement of callbacks and errbacks.

Summary

After thinking some more about callbacks, we realize that letting callback exceptions bubble up the stack isn't going to work out so well, since callback programming inverts the usual relationship between low-context and high-context code. And the Deferred class tackles this problem by catching exceptions and sending them down the chain instead of up into the reactor.

We've also learned that ordinary results (`return` values) move down the chain as well. Combining both facts together results in a kind of criss-cross firing pattern as the deferred switches back and forth between the callback and errback lines, depending on the result of each stage.

Armed with this knowledge, in Part 10 we will update our poetry client with some poetry transformation logic.

Suggested Exercises

1. Inspect the implementation of each of the four methods on the `Deferred` which add callbacks and errbacks. Verify that all methods add a callback/errback pair.
2. Use the deferred simulator to investigate the difference between this code:

   ```
   1 | deferred.addCallbacks(my_callback, my_errback)
   ```

   and this code:

   ```
   1 | deferred.addCallback(my_callback)
   2 | deferred.addErrback(my_errback)
   ```

   Recall that the last two methods add implicit pass-through functions as one member of the pair.

Part 10: Poetry Transformed

This continues the introduction started here. You can find an index to the entire series here.

Client 5.0

Now we're going to add some transformation logic to our poetry client, along the lines suggested in Part 9. But first, I have a shameful and humbling confession to make: I don't know how to write the Byronification Engine. It is beyond my programming abilities. So instead, I'm going to implement a simpler transformation, the Cummingsifier. The Cummingsifier is an algorithm that takes a poem and returns a new poem like the original but written in the style of e.e. cummings. Here is the Cummingsifier algorithm in its entirety:

```
1 | def cummingsify(poem)
2 |     return poem.lower()
```

Unfortunately, this algorithm is so simple it never actually fails, so in client 5.0, located in `twisted-client-5/get-poetry.py`, we use a modified version of `cummingsify` that randomly does one of the following:

1. Return a cummingsified version of the poem.

2. Raise a GibberishError.
3. Raise a ValueError.

In this way we simulate a more complicated algorithm that sometimes fails in unexpected ways.

The only other changes in client 5.0 are in the `poetry_main` function:

```
 1  def poetry_main():
 2      addresses = parse_args()
 3
 4      from twisted.internet import reactor
 5
 6      poems = []
 7      errors = []
 8
 9      def try_to_cummingsify(poem):
10          try:
11              return cummingsify(poem)
12          except GibberishError:
13              raise
14          except:
15              print 'Cummingsify failed!'
16              return poem
17
18      def got_poem(poem):
19          print poem
20          poems.append(poem)
21
22      def poem_failed(err):
23          print >>sys.stderr, 'The poem download failed.'
24          errors.append(err)
25
26      def poem_done(_):
27          if len(poems) + len(errors) == len(addresses):
28              reactor.stop()
29
30      for address in addresses:
31          host, port = address
32          d = get_poetry(host, port)
33          d.addCallback(try_to_cummingsify)
34          d.addCallbacks(got_poem, poem_failed)
35          d.addBoth(poem_done)
36
37      reactor.run()
```

So when the program downloads a poem from the server, it will either:

1. Print the cummingsified (lower-cased) version of the poem.
2. Print "Cummingsify failed!" followed by the original poem.
3. Print "The poem download failed."

Although we have retained the ability to download from multiple servers, when you are testing out client 5.0 it's easier to just use a single server and run the program multiple times, until you see all three different outcomes. Also try running the client on a port with no server.

Let's draw the callback/errback chain we create on each `Deferred` we get back from `get_poetry`:

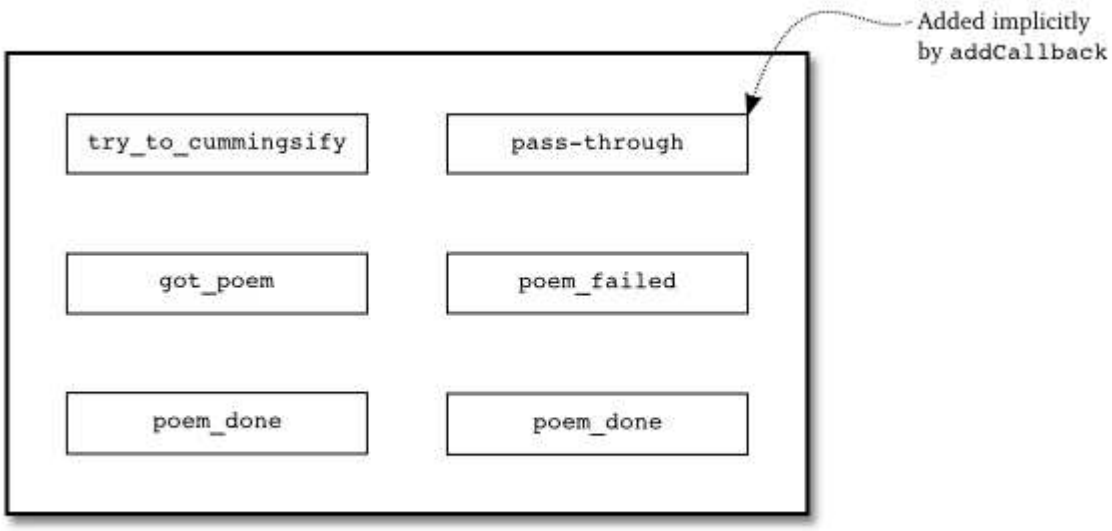

Figure 19: the deferred chain in client 5.0

Note the pass-through errback that gets added by `addCallback`. It passes whatever `Failure` it receives onto the next errback (`poem_failed`). Thus, `poem_failed` can handle failures from both `get_poetry` (i.e., the deferred is fired with the `errback` method) and the `cummingsify` function.

Also note the hauntingly beautiful drop-shadow around the border of the deferred in Figure 19. It doesn't signify anything other than me discovering how to do it in Inkscape. Expect more drop-shadows in the future.

Let's analyze the different ways our deferred can fire. The case where we get a poem and the `cummingsify` function works correctly is shown in Figure 20:

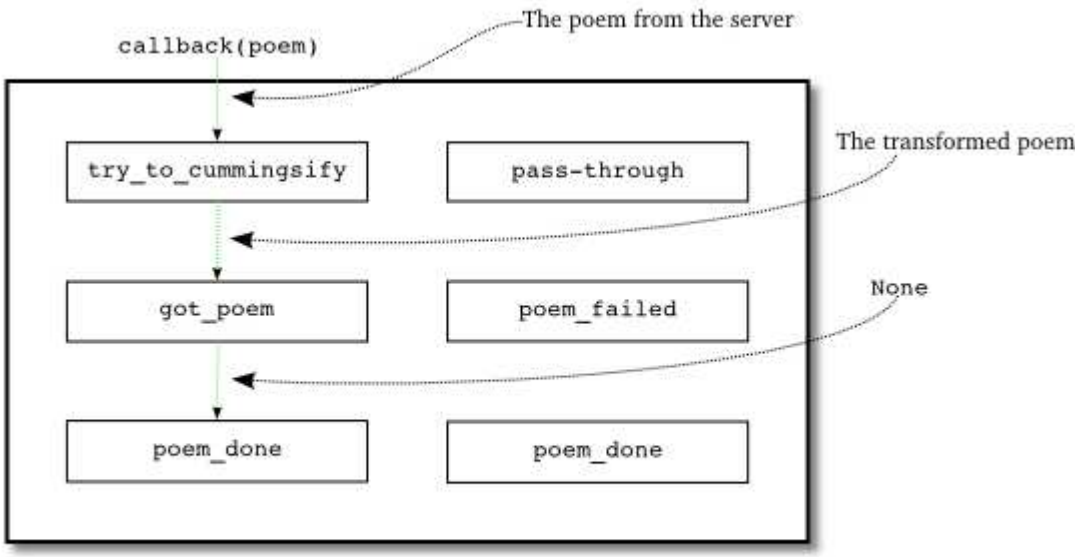

Figure 20: when we download a poem and transform it correctly

In this case no callback fails, so control flows down the callback line. Note that `poem_done` receives `None`

as its result, since `got_poem` doesn't actually `return` a value. If we wanted subsequent callbacks to have access to the poem, we would modify `got_poem` to return the poem explicitly.

Figure 21 shows the case where we get a poem, but `cummingsify` raises a GibberishError:

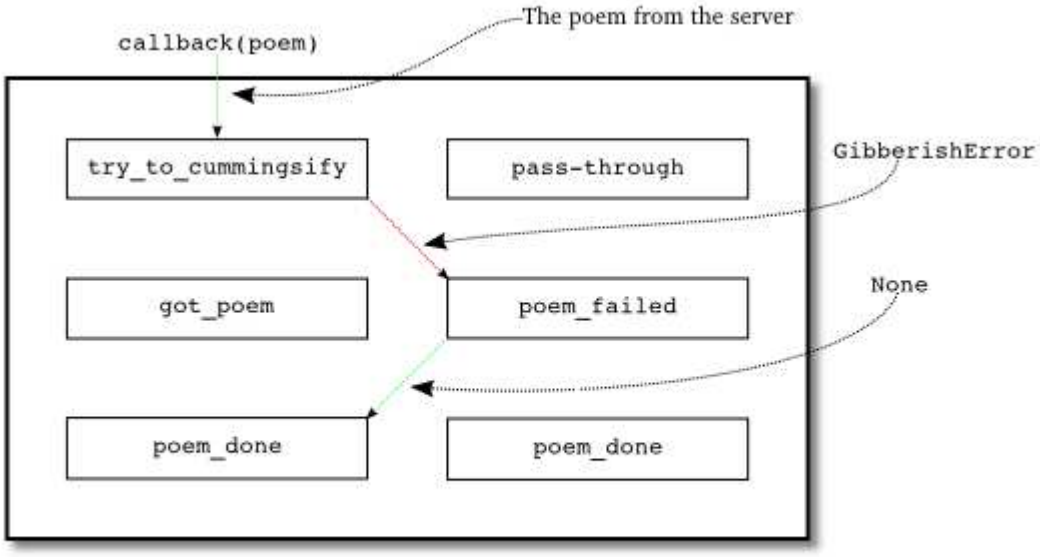

Figure 21: when we download a poem and get a GibberishError

Since the `try_to_cummingsify` callback re-raises a `GibberishError`, control switches to the errback line and `poem_failed` is called with the exception as its argument (wrapped in a `Failure`, of course).

And since `poem_failed` doesn't raise an exception, or return a `Failure`, after it is done control switches back to the callback line. If we want `poem_failed` to handle the error completely, then returning `None` is a reasonable behavior. On the other hand, if we wanted `poem_failed` to take some action, but still propagate the error, we could change `poem_failed` to return its `err` argument and processing would continue down the errback line.

Note that in the current code neither `got_poem` nor `poem_failed` ever fail themselves, so the `poem_done` errback will never be called. But it's safe to add it in any case and doing so represents an instance of "defensive" programming, as either `got_poem` or `poem_failed` might have bugs we don't know about. Since the `addBoth` method ensures that a particular function will run no matter how the deferred fires, using `addBoth` is analogous to adding a `finally` clause to a `try/except` statement.

Now examine the case where we download a poem and the `cummingsify` function raises a `ValueError`, displayed in Figure 22:

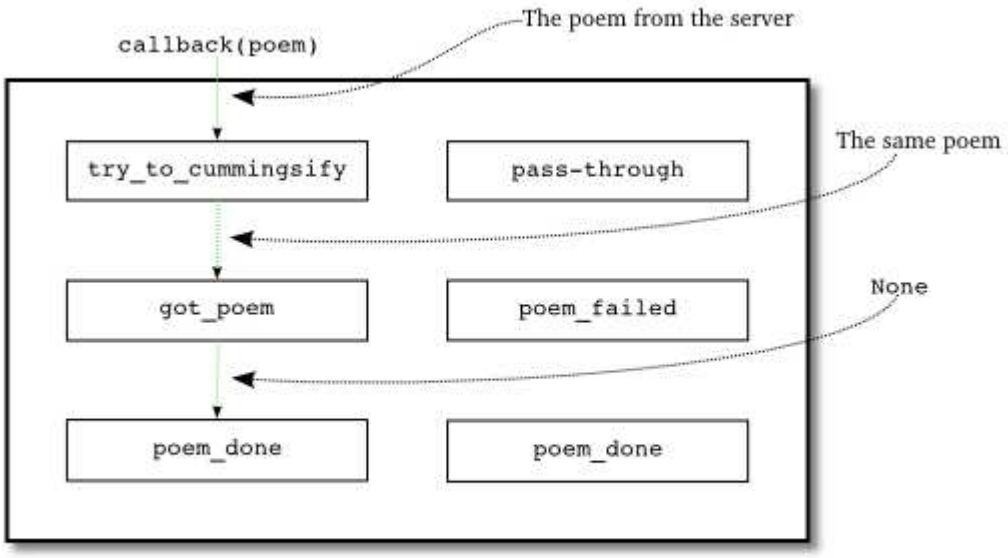

Figure 22: when we download a poem and cummingsify fails

This is the same as figure 20, except `got_poem` receives the original version of the poem instead of the transformed version. The switch happens entirely inside the `try_to_cummingsify` callback, which traps the `ValueError` with an ordinary `try/except` statement and returns the original poem instead. The deferred object never sees that error at all.

Lastly, we show the case where we try to download a poem from a non-existent server in Figure 23:

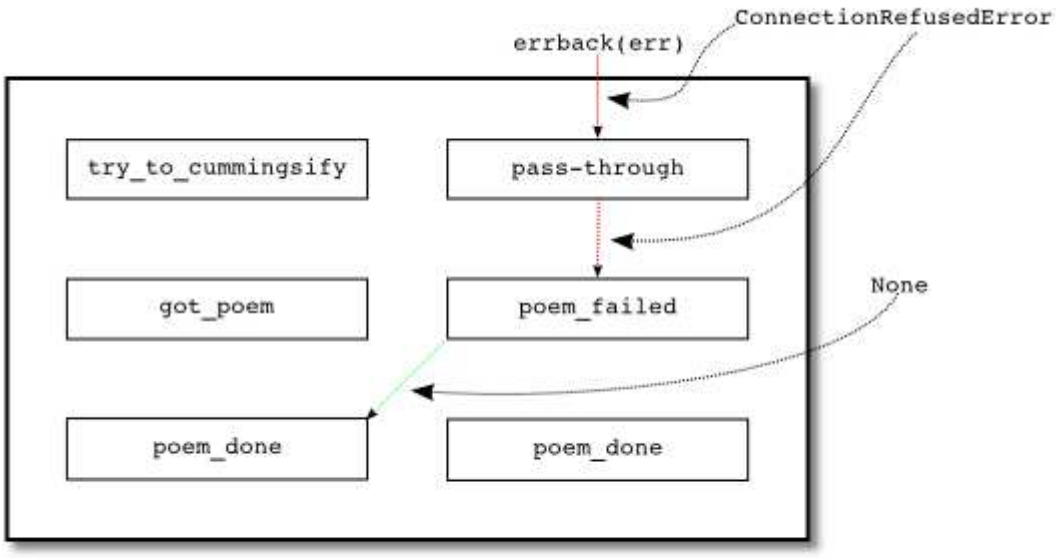

Figure 23: when we cannot connect to a server

As before, `poem_failed` returns `None` so afterwards control switches to the callback line.

Client 5.1

In client 5.0 we are trapping exceptions from `cummingsify` in our `try_to_cummingsify` callback using an ordinary `try`/`except` statement, rather than letting the deferred catch them first. There isn't necessarily anything wrong with this strategy, but it's instructive to see how we might do this differently.

Let's suppose we wanted to let the deferred catch both `GibberishError` and `ValueError` exceptions and send them to the errback line. To preserve the current behavior our subsequent errback needs to check to see if the error is a `ValueError` and, if so, handle it by returning the original poem, so that control goes back to the callback line and the original poem gets printed out.

But there's a problem: the errback wouldn't get the original poem, it would get the `Failure`-wrapped `ValueError` raised by the `cummingsify` function. To let the errback handle the error, we need to arrange for it to receive the original poem.

One way to do that is to modify the `cummingsify` function so the original poem is included in the exception. That's what we've done in client 5.1, located in twisted-client-5/get-poetry-1.py. We changed the `ValueError` exception into a custom `CannotCummingsify` exception which takes the original poem as the first argument.

If `cummingsify` were a real function in an external module, then it would probably be best to wrap it with another function that trapped any exception that wasn't `GibberishError` and raise a `CannotCummingsify` exception instead. With this new setup, our `poetry_main` function looks like this:

```
 1  def poetry_main():
 2      addresses = parse_args()
 3
 4      from twisted.internet import reactor
 5
 6      poems = []
 7      errors = []
 8
 9      def cummingsify_failed(err):
10          if err.check(CannotCummingsify):
11              print 'Cummingsify failed!'
12              return err.value.args[0]
13          return err
14
15      def got_poem(poem):
16          print poem
17          poems.append(poem)
18
19      def poem_failed(err):
20          print >>sys.stderr, 'The poem download failed.'
21          errors.append(err)
22
23      def poem_done(_):
24          if len(poems) + len(errors) == len(addresses):
25              reactor.stop()
26
27      for address in addresses:
28          host, port = address
29          d = get_poetry(host, port)
30          d.addCallback(cummingsify)
31          d.addErrback(cummingsify_failed)
32          d.addCallbacks(got_poem, poem_failed)
33          d.addBoth(poem_done)
```

And each deferred we create has the structure pictured in Figure 24:

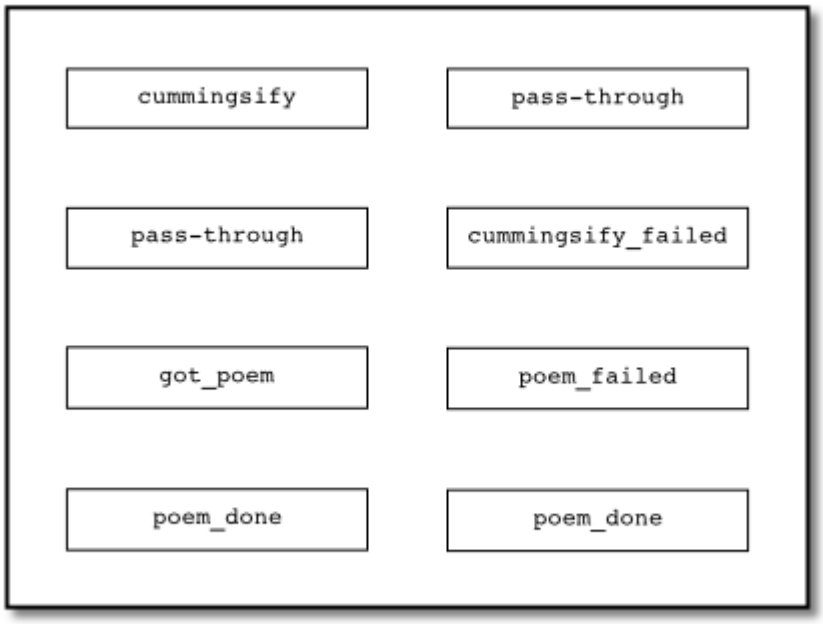

Figure 24: the deferred chain in client 5.1

Examine the `cummingsify_failed` errback:

```
1  def cummingsify_failed(err):
2      if err.check(CannotCummingsify):
3          print 'Cummingsify failed!'
4          return err.value.args[0]
5      return err
```

We are using the `check` method on `Failure` objects to test whether the exception embedded in the `Failure` is an instance of `CannotCummingsify`. If so, we return the first argument to the exception (the original poem) and thus handle the error. Since the return value is not a `Failure`, control returns to the callback line. Otherwise, we return the `Failure` itself and send (re-raise) the error down the errback line. As you can see, the exception is available as the `value` attribute on the `Failure`.

Figure 25 shows what happens when we get a `CannotCummingsify` exception:

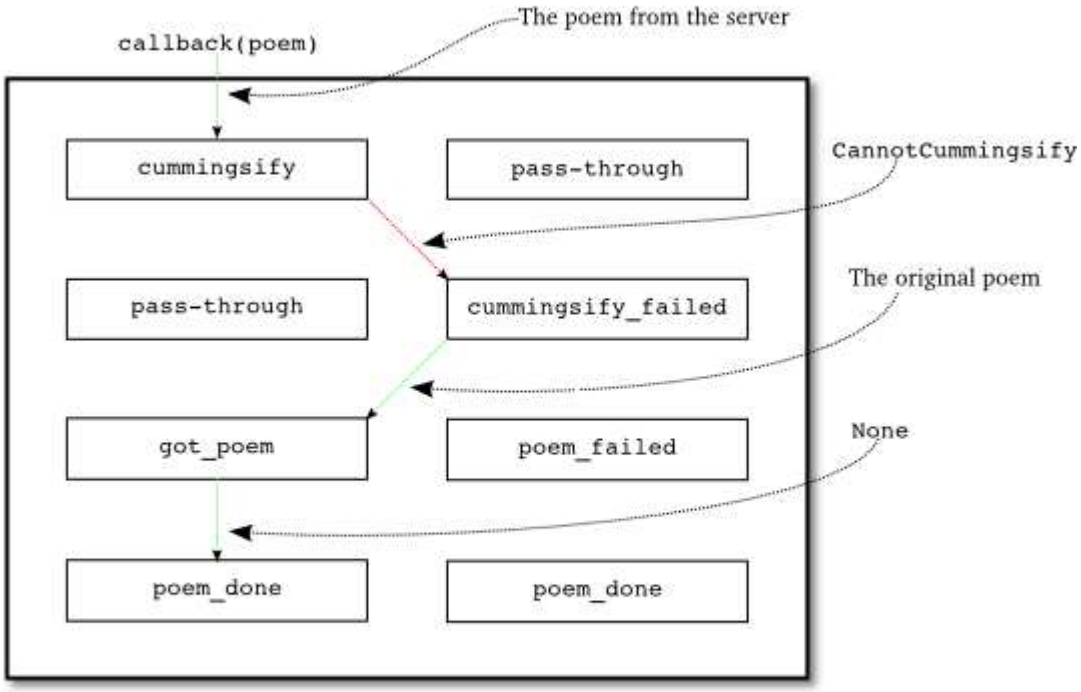

Figure 25: when we get a CannotCummingsify error

So when we are using a deferred, we can sometimes choose whether we want to use `try/except` statements to handle exceptions, or let the deferred re-route errors to an errback.

Summary

In Part 10 we updated our poetry client to make use of the `Deferred`'s ability to route errors and results down the chain. Although the example was rather artificial, it did illustrate how control flow in a deferred switches back and forth between the callback and errback line depending on the result of each stage.

So now we know everything there is to know about deferreds, right? Not yet! We're going to explore some more features of deferreds in a future Part. But first we'll take a little detour and, in Part 11, implement a Twisted version of our poetry server.

Suggested Exercises

1. Figure 25 shows one of the four possible ways the deferreds in client 5.1 can fire. Draw the other three.
2. Use the deferred simulator to simulate all possible firings for clients 5.0 and 5.1. To get you started, this simulator program can represent the case where the `try_to_cummingsify` function succeeds in client 5.0:

```
r poem p
r None r None
r None r None
```

Part 11: Your Poetry is Served

This continues the introduction started here. You can find an index to the entire series here.

A Twisted Poetry Server

Now that we've learned so much about writing clients with Twisted, let's turn around and re-implement our poetry server with Twisted too. And thanks to the generality of Twisted's abstractions, it turns out we've already learned almost everything we need to know. Take a look at our Twisted poetry server located in `twisted-server-1/fastpoetry.py`. It's called `fastpoetry` because this server sends the poetry as fast as possible, without any delays at all. Note there's significantly less code than in the client!

Let's take the pieces of the server one at a time. First, the `PoetryProtocol`:

```
1  class PoetryProtocol(Protocol):
2
3      def connectionMade(self):
4          self.transport.write(self.factory.poem)
5          self.transport.loseConnection()
```

Like the client, the server uses a `Protocol` instance to manage connections (in this case, connections that clients make to the server). Here the `Protocol` is implementing the server-side portion of our poetry protocol. Since our wire protocol is strictly one-way, the server's `Protocol` instance only needs to be concerned with sending data. If you recall, our wire protocol requires the server to start sending the poem immediately after the connection is made, so we implement the `connectionMade` method, a callback that is invoked after a `Protocol` instance is connected to a `Transport`.

Our method tells the `Transport` to do two things: send the entire text of the poem (`self.transport.write`) and close the connection (`self.transport.loseConnection`). Of course, both of those operations are asynchronous. So the call to `write()` really means "eventually send all this data to the client" and the call to `loseConnection()` really means "close this connection once all the data I've asked you to write has been written".

As you can see, the `Protocol` retrieves the text of the poem from the `Factory`, so let's look at that next:

```
1  class PoetryFactory(ServerFactory):
2
3      protocol = PoetryProtocol
4
5      def __init__(self, poem):
6          self.poem = poem
```

Now that's pretty darn simple. Our factory's only real job, besides making `PoetryProtocol` instances on demand, is storing the poem that each `PoetryProtocol` sends to a client.

Notice that we are sub-classing `ServerFactory` instead of `ClientFactory`. Since our server is passively listening for connections instead of actively making them, we don't need the extra methods `ClientFactory` provides. How can we be sure of that? Because we are using the `listenTCP` reactor method and the documentation for that method explains that the `factory` argument should be an instance of `ServerFactory`.

Here's the `main` function where we call `listenTCP`:

```
1  def main():
2      options, poetry_file = parse_args()
3
4      poem = open(poetry_file).read()
5
6      factory = PoetryFactory(poem)
7
8      from twisted.internet import reactor
```

```
 9
10      port = reactor.listenTCP(options.port or 0, factory,
11                               interface=options.iface)
12
13      print 'Serving %s on %s.' % (poetry_file, port.getHost())
14
15      reactor.run()
```

It basically does three things:

1. Read the text of the poem we are going to serve.
2. Create a `PoetryFactory` with that poem.
3. Use `listenTCP` to tell Twisted to listen for connections on a port, and use our factory to make the protocol instances for each new connection.

After that, the only thing left to do is tell the `reactor` to start running the loop. You can use any of our previous poetry clients (or just `netcat`) to test out the server.

Discussion

Recall Figure 8 and Figure 9 from Part 5. Those figures illustrated how a new `Protocol` instance is created and initialized after Twisted makes a new connection on our behalf. It turns out the same mechanism is used when Twisted accepts a new incoming connection on a port we are listening on. That's why both `connectTCP` and `listenTCP` require `factory` arguments.

One thing we didn't show in Figure 9 is that the `connectionMade` callback is also called as part of `Protocol` initialization. This happens no matter what, but we didn't need to use it in the client code. And the `Protocol` methods that we did use in the client aren't used in the server's implementation. So if we wanted to, we could make a shared library with a single `PoetryProtocol` that works for both clients and servers. That's actually the way things are typically done in Twisted itself. For example, the `NetstringReceiver` `Protocol` can both read and write netstrings from and to a `Transport`.

We skipped writing a low-level version of our server, but let's think about what sort of things are going on under the hood. First, calling `listenTCP` tells Twisted to create a listening socket and add it to the event loop. An "event" on a listening socket doesn't mean there is data to read; instead it means there is a client waiting to connect to us.

Twisted will automatically accept incoming connection requests, thus creating a new client socket that links the server directly to an individual client. That client socket is also added to the event loop, and Twisted creates a new `Transport` and (via the `PoetryFactory`) a new `PoetryProtocol` instance to service that specific client. So the `Protocol` instances are always connected to client sockets, never to the listening socket.

We can visualize all of this in Figure 26:

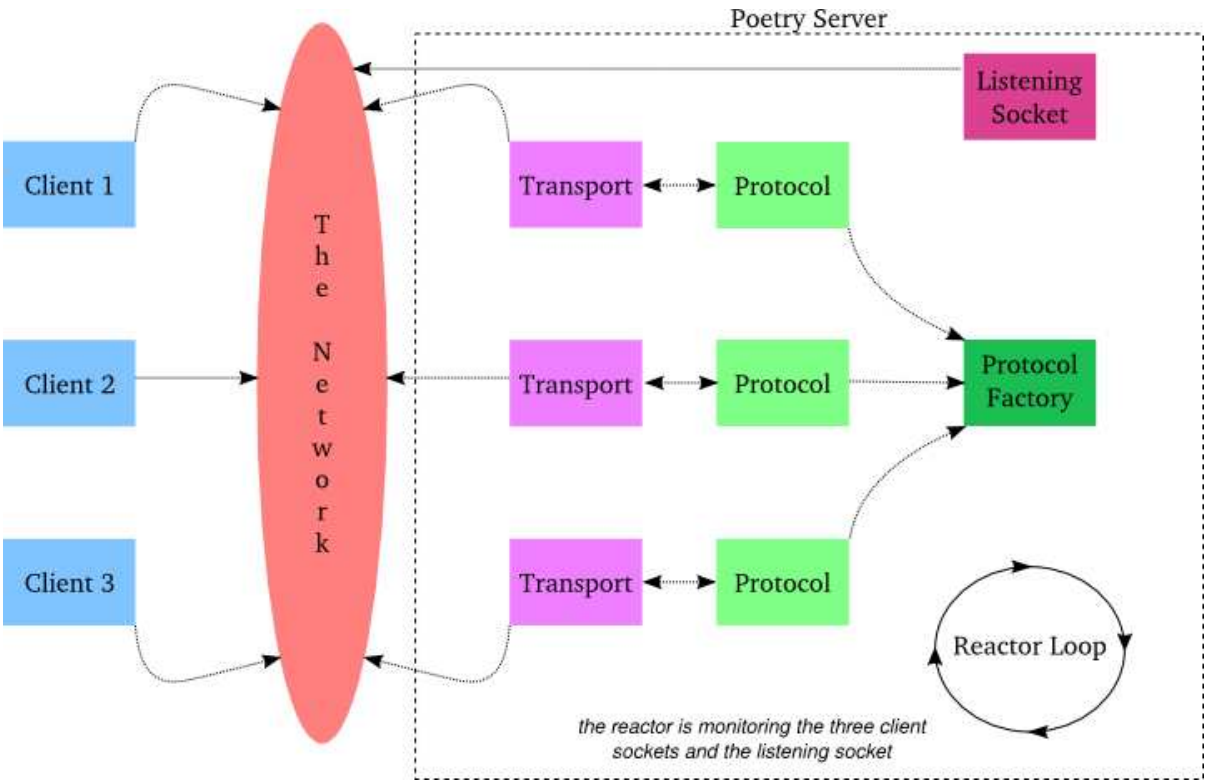

Figure 26: the poetry server in action

In the figure there are three clients currently connected to the poetry server. Each `Transport` represents a single client socket, and the listening socket makes a total of four file descriptors for the select loop to monitor. When a client is disconnected the associated `Transport` and `PoetryProtocol` will be dereferenced and garbage-collected (assuming we haven't stashed a reference to one of them somewhere, a practice we should avoid to prevent memory leaks). The `PoetryFactory`, meanwhile, will stick around as long as we keep listening for new connections which, in our poetry server, is forever. Like the beauty of poetry. Or something. At any rate, Figure 26 certainly cuts a fine figure of a Figure, doesn't it?

The client sockets and their associated Python objects won't live very long if the poem we are serving is relatively short. But with a large poem and a really busy poetry server we could end up with hundreds or thousands of simultaneous clients. And that's OK — Twisted has no built-in limits on the number of connections it can handle. Of course, as you increase the load on any server, at some point you will find it cannot keep up or some internal OS limit is reached. For highly-loaded servers, careful measurement and testing is the order of the day.

Twisted also imposes no limit on the number of ports we can listen on. In fact, a single Twisted process could listen on dozens of ports and provide a different service on each one (by using a different factory class for each `listenTCP` call). And with careful design, whether you provide multiple services with a single Twisted process or several is a decision you could potentially even postpone to the deployment phase.

There's a couple things our server is missing. First of all, it doesn't generate any logs that might help us debug problems or analyze our network traffic. Furthermore, the server doesn't run as a daemon, making it vulnerable to death by accidental `Ctrl-C` (or just logging out). We'll fix both those problems in a future Part but first, in Part 12, we'll write another server to perform poetry transformation.

Suggested Exercises

1. Write an asynchronous poetry server without using Twisted, like we did for the client in Part 2. Note that listening sockets need to be monitored for reading and a "readable" listening socket means we can `accept` a new client socket.

2. Write a low-level asynchronous poetry server using Twisted, but without using `listenTCP` or protocols, transports, and factories, like we did for the client in Part 4. So you'll still be making your own sockets, but you can use the Twisted reactor instead of your own `select` loop.
3. Make the high-level version of the Twisted poetry server a "slow server" by using `callLater` or `LoopingCall` to make multiple calls to `transport.write()`. Add the `--num-bytes` and `--delay` command line options supported by the blocking server. Don't forget to handle the case where the client disconnects before receiving the whole poem.
4. Extend the high-level Twisted server so it can serve multiple poems (on different ports).
5. What are some reasons to serve multiple services from the same Twisted process? What are some reasons not to?

Part 12: A Poetry Transformation Server

This continues the introduction started here. You can find an index to the entire series here.

One More Server

Alright, we've written one Twisted server so let's write another, and then we'll get back to learning some more about Deferreds.

In Parts 9 and 10 we introduced the idea of a poetry transformation engine. The one we eventually implemented, the cummingsifier, was so simple we had to add random exceptions to simulate a failure. But if the transformation engine was located on another server, providing a network "poetry transformation service", then there is a much more realistic failure mode: the transformation server is down.

So in Part 12 we're going to implement a poetry transformation server and then, in the next Part, we'll update our poetry client to use the external transformation service and learn a few new things about Deferreds in the process.

Designing the Protocol

Up till now the interactions between client and server have been strictly one-way. The server sends a poem to the client while the client never sends anything at all to the server. But a transformation service is two-way — the client sends a poem to the server and then the server sends a transformed poem back. So we'll need to use, or invent, a protocol to handle that interaction.

While we're at it, let's allow the server to support multiple kinds of transformations and allow the client to select which one to use. So the client will send two pieces of information: the name of the transformation and the complete text of the poem. And the server will return a single piece of information, namely the text of the transformed poem. So we've got a very simple sort of Remote Procedure Call.

Twisted includes support for several protocols we could use to solve this problem, including XML-RPC, Perspective Broker, and AMP.

But introducing any of these full-featured protocols would require us to go too far afield, so we'll roll our own humble protocol instead. Let's have the client send a string of the form (without the angle brackets):

<transform-name>.<text of the poem>

That's just the name of the transform, followed by a period, followed by the complete text of the poem itself. And we'll encode the whole thing in the form of a netstring. And the server will send back the text of the transformed poem, also in a netstring. Since netstrings use length-encoding, the client will be able to detect the case where the server fails to send back a complete result (maybe it crashed in the middle of

the operation). If you recall, our original poetry protocol has trouble detecting aborted poetry deliveries.

So much for the protocol design. It's not going to win any awards, but it's good enough for our purposes.

The Code

Let's look at the code of our transformation server, located in [twisted-server-1/transformedpoetry.py](). First, we define a `TransformService` class:

```
1  class TransformService(object):
2
3      def cummingsify(self, poem):
4          return poem.lower()
```

The transform service currently implements one transformation, `cummingsify`, via a method of the same name. We could add additional algorithms by adding additional methods. Here's something important to notice: the transformation service is entirely independent of the particular details of the protocol we settled on earlier. Separating the protocol logic from the service logic is a common pattern in Twisted programming. Doing so makes it easy to provide the same service via multiple protocols without duplicating code.

Now let's look at the protocol factory (we'll look at the protocol right after):

```
 1  class TransformFactory(ServerFactory):
 2
 3      protocol = TransformProtocol
 4
 5      def __init__(self, service):
 6          self.service = service
 7
 8      def transform(self, xform_name, poem):
 9          thunk = getattr(self, 'xform_%s' % (xform_name,), None)
10
11          if thunk is None: # no such transform
12              return None
13
14          try:
15              return thunk(poem)
16          except:
17              return None # transform failed
18
19      def xform_cummingsify(self, poem):
20          return self.service.cummingsify(poem)
```

This factory provides a `transform` method which a protocol instance can use to request a poetry transformation on behalf of a connected client. The method returns `None` if there is no such transformation or if the transformation fails. And like the `TransformService`, the protocol factory is independent of the wire-level protocol, the details of which are delegated to the protocol class itself.

One thing to notice is the way we guard access to the service though the `xform_`-prefixed methods. This is a pattern you will find in the Twisted sources, although the prefixes vary and they are usually on an object separate from the factory. It's one way of preventing client code from executing an arbitrary method on the service object, since the client can send any transform name they want. It also provides a place to perform protocol-specific adaptation to the API provided by the service object.

Now we'll take a look at the protocol implementation:

```
1  class TransformProtocol(NetstringReceiver):
2
3      def stringReceived(self, request):
```

```
4           if '.' not in request: # bad request
5               self.transport.loseConnection()
6               return
7
8           xform_name, poem = request.split('.', 1)
9
10          self.xformRequestReceived(xform_name, poem)
11
12      def xformRequestReceived(self, xform_name, poem):
13          new_poem = self.factory.transform(xform_name, poem)
14
15          if new_poem is not None:
16              self.sendString(new_poem)
17
18          self.transport.loseConnection()
```

In the protocol implementation we take advantage of the fact that Twisted supports netstrings via the `NetstringReceiver` protocol. That base class takes care of decoding (and encoding) the netstrings and all we have to do is implement the `stringReceived` method. In other words, `stringReceived` is called with the *content* of a netstring sent by the client, without the extra bytes added by the netstring encoding. The base class also takes care of buffering the incoming bytes until we have enough to decode a complete string.

If everything goes ok (and if it doesn't we just close the connection) we send the transformed poem back to the client using the `sendString` method provided by `NetstringReceiver` (and which ultimately calls `transport.write()`). And that's all there is to it. We won't bother listing the `main` function since it's similar to the ones we've seen before.

Notice how we continue the Twisted pattern of translating the incoming byte stream to higher and higher levels of abstraction by defining the `xformRequestReceived` method, which is passed the name of the transform and the poem as two separate arguments.

A Simple Client

We'll implement a Twisted client for the transformation service in the next Part. For now we'll just make do with a simple script located in `twisted-server-1/transform-test`. It uses the netcat program to send a poem to the server and then prints out the response (which will be encoded as a netstring). Let's say you run the transformation server on port 11000 like this:

```
python twisted-server-1/transformedpoetry.py --port 11000
```

Then you could run the test script against that server like this:

```
./twisted-server-1/transform-test 11000
```

And you should see some output like this:

```
15:here is my poem,
```

That's the netstring-encoded transformed poem (the original is in all upper case).

Discussion

We introduced a few new ideas in this Part:

1. Two-way communication.
2. Building on an existing protocol implementation provided by Twisted.
3. Using a service object to separate functional logic from protocol logic.

The basic mechanics of two-way communication are simple. We used the same techniques for reading and writing data in previous clients and servers; the only difference is we used them both together. Of course, a more complex protocol will require more complex code to process the byte stream and format outgoing messages. And that's a great reason to use an existing protocol implementation like we did today.

Once you start getting comfortable writing basic protocols, it's a good idea to take a look at the different protocol implementations provided by Twisted. You might start by perusing the `twisted.protocols.basic` module and going from there. Writing simple protocols is a great way to familiarize yourself with the Twisted style of programming, but in a "real" program it's probably a lot more common to use a ready-made implementation, assuming there is one available for the protocol you want to use.

The last new idea we introduced, the use of a Service object to separate functional and protocol logic, is a really important design pattern in Twisted programming. Although the service object we made today is trivial, you can imagine a more realistic network service could be quite complex. And by making the Service independent of protocol-level details, we can quickly provide the same service on a new protocol without duplicating code.

Figure 27 shows a transformation server that is providing poetry transformations via two different protocols (the version of the server we presented above only has one protocol):

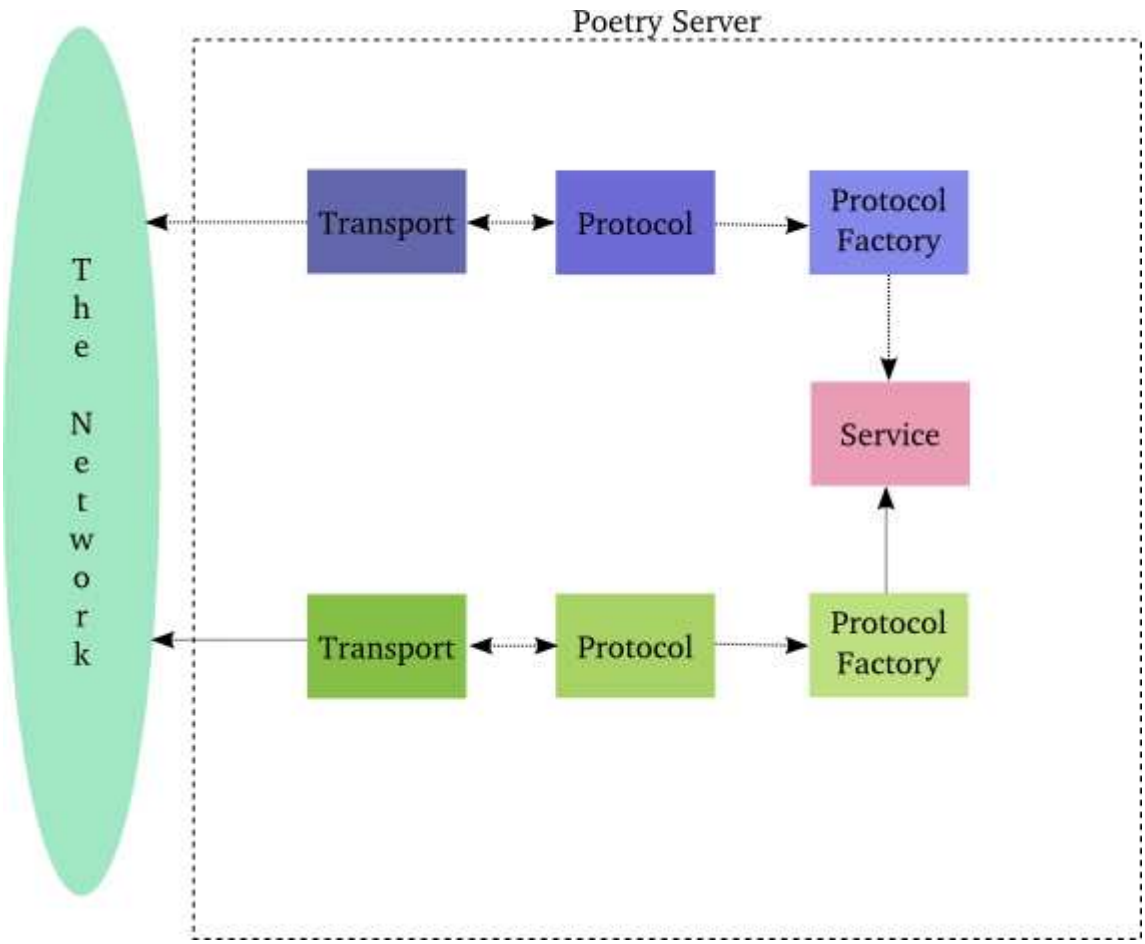

Figure 27: a transformation server with two protocols

Although we need two separate protocol factories in Figure 27, they might differ only in their `protocol` class attribute and would be otherwise identical. The factories would share the same Service object and only the `Protocol`s themselves would require separate implementations. Now that's code re-use!

Looking Ahead

So much for our transformation server. In Part 13, we'll update our poetry client to use the transform server instead of implementing transformations in the client itself.

Suggested Exercises

1. Read the source code for the `NetstringReceiver` class. What happens if the client sends a malformed netstring? What happens if the client tries to send a huge netstring?
2. Invent another transformation algorithm and add it to the transformation service and the protocol factory. Test it out by modifying the netcat client.
3. Invent another protocol for requesting poetry transformations and modify the server to handle both protocols (on two different ports). Use the same instance of the `TransformService` for both.
4. How would the code need to change if the methods on the `TransformService` were asynchronous (i.e., they returned Deferreds)?
5. Write a synchronous client for the transformation server.
6. Update the original client and server to use netstrings when sending poetry.

Part 13: Deferred All The Way Down

This continues the introduction started here. You can find an index to the entire series here.

Introduction

Recall poetry client 5.1 from Part 10.The client used a Deferred to manage a callback chain that included a call to a poetry transformation engine. In client 5.1, the engine was implemented as a synchronous function call implemented in the client itself.

Now we want to make a new client that uses the networked poetry transformation service we wrote in Part 12. But here's the wrinkle: since the transformation service is accessed over the network, we'll need to use asynchronous I/O. And that means our API for requesting a transformation will have to be asynchronous, too. In other words, the `try_to_cummingsify` callback is going to return a `Deferred` in our new client.

So what happens when a callback in a deferred's chain returns another deferred? Let's call the first deferred the 'outer' deferred and the second the 'inner' one. Suppose callback **N** in the outer deferred returns the inner deferred. That callback is saying "I'm asynchronous, my result isn't here yet". Since the outer deferred needs to call the next callback or errback in the chain with the result, the outer deferred needs to wait until the inner deferred is fired. Of course, the outer deferred can't block either, so instead the outer deferred suspends the execution of the callback chain and returns control to the reactor (or whatever fired the outer deferred).

And how does the outer deferred know when to resume? Simple — by adding a callback/errback pair to the inner deferred. Thus, when the inner deferred is fired the outer deferred will resume executing its chain. If the inner deferred succeeds (i.e., it calls the callback added by the outer deferred), then the outer deferred calls its **N+1** callback with the result. And if the inner deferred fails (calls the errback added by the outer deferred), the outer deferred calls the **N+1** errback with the failure.

That's a lot to digest, so let's illustrate the idea in Figure 28:

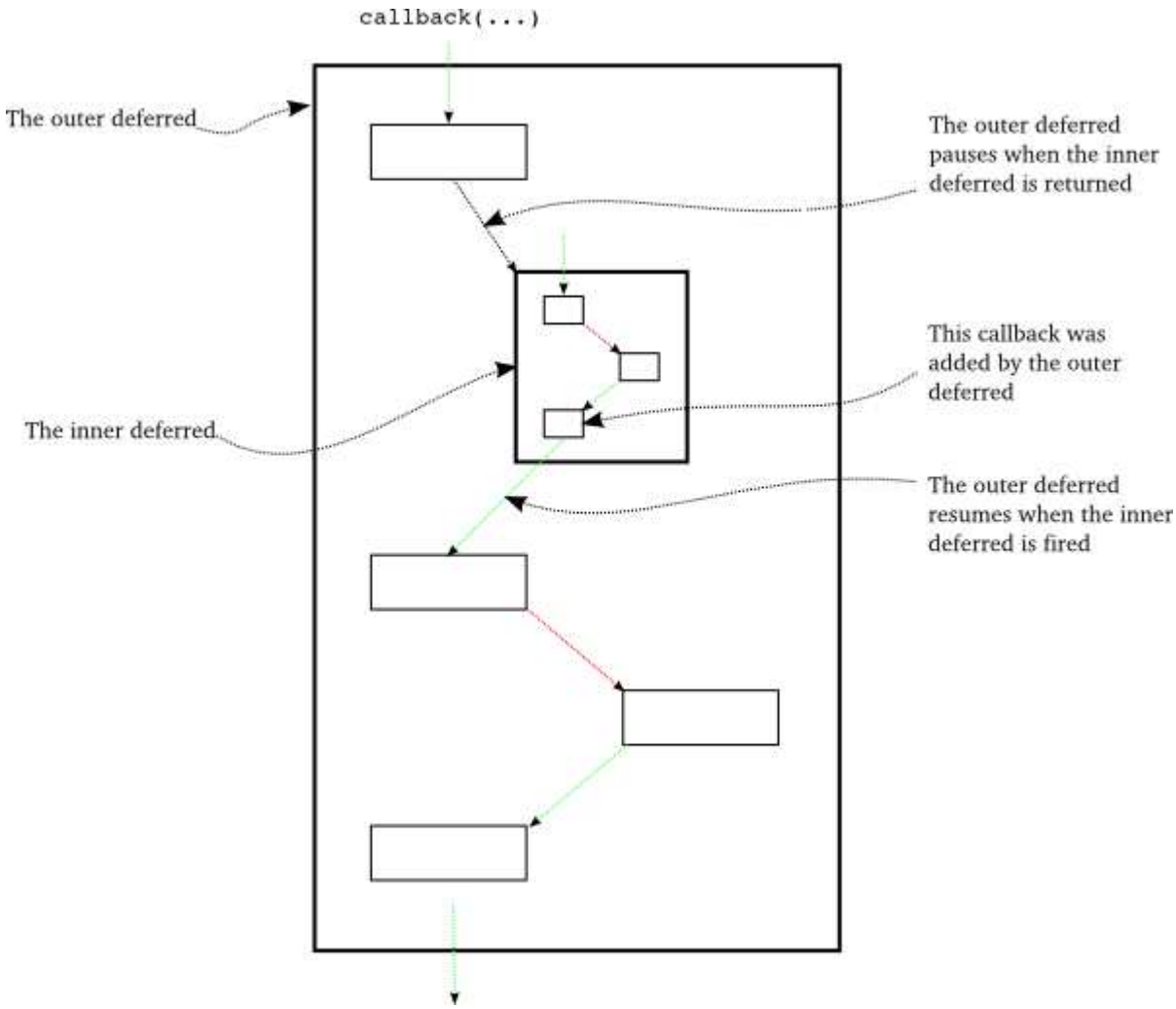

Figure 28: outer and inner deferred processing

In this figure the outer deferred has 4 layers of callback/errback pairs. When the outer deferred fires, the first callback in the chain returns a deferred (the inner deferred). At that point, the outer deferred will stop firing its chain and return control to the reactor (after adding a callback/errback pair to the inner deferred). Then, some time later, the inner deferred fires and the outer deferred resumes processing its callback chain. Note the outer deferred does *not* fire the inner deferred itself. That would be impossible, since the outer deferred cannot know when the inner deferred's result is available, or what that result might be. Rather, the outer deferred simply waits (asynchronously) for the inner deferred to fire.

Notice how the line connecting the callback to the inner deferred in Figure 28 is black instead of green or red. That's because we don't know whether the callback succeeded or failed until the inner deferred is fired. Only then can the outer deferred decide whether to call the next callback or the next errback in its own chain.

Figure 29 shows the same outer/inner deferred firing sequence in Figure 28 from the point of view of the reactor:

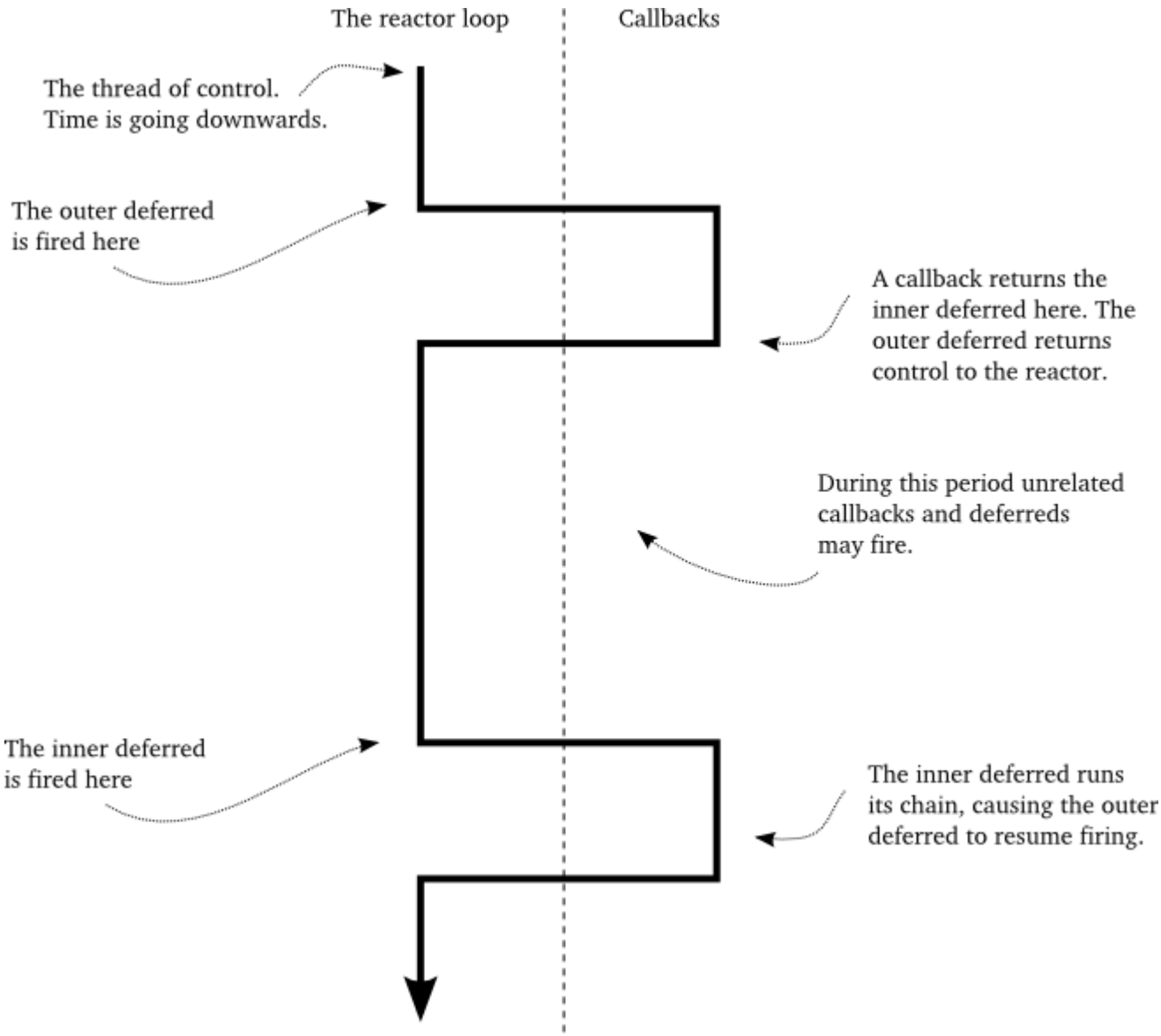

Figure 29: the thread of control in Figure 28

This is probably the most complicated feature of the `Deferred` class, so don't worry if you need some time to absorb it. We'll illustrate it one more way using the example code in `twisted-deferred/defer-10.py`. That example creates two outer deferreds, one with plain callbacks, and one where a single callback returns an inner deferred. By studying the code and the output you can see how the second outer deferred stops running its chain when the inner deferred is returned, and then starts up again when the inner deferred is fired.

Client 6.0

Let's use our new knowledge of nested deferreds and re-implement our poetry client to use the network transformation service from Part 12. You can find the code in `twisted-client-6/get-poetry.py`. The poetry Protocol and Factory are unchanged from the previous version. But now we have a Protocol and Factory for making transformation requests. Here's the transform client Protocol:

```
1  class TransformClientProtocol(NetstringReceiver):
2
3      def connectionMade(self):
4          self.sendRequest(self.factory.xform_name, self.factory.poem)
5
6      def sendRequest(self, xform_name, poem):
7          self.sendString(xform_name + '.' + poem)
8
```

```
 9       def stringReceived(self, s):
10           self.transport.loseConnection()
11           self.poemReceived(s)
12
13       def poemReceived(self, poem):
14           self.factory.handlePoem(poem)
```

Using the NetstringReceiver as a base class makes this implementation pretty simple. As soon as the connection is established we send the transform request to the server, retrieving the name of the transform and the poem from our factory. And when we get the poem back, we pass it on to the factory for processing. Here's the code for the Factory:

```
 1  class TransformClientFactory(ClientFactory):
 2
 3      protocol = TransformClientProtocol
 4
 5      def __init__(self, xform_name, poem):
 6          self.xform_name = xform_name
 7          self.poem = poem
 8          self.deferred = defer.Deferred()
 9
10      def handlePoem(self, poem):
11          d, self.deferred = self.deferred, None
12          d.callback(poem)
13
14      def clientConnectionLost(self, _, reason):
15          if self.deferred is not None:
16              d, self.deferred = self.deferred, None
17              d.errback(reason)
18
19      clientConnectionFailed = clientConnectionLost
```

This factory is designed for clients and handles a single transformation request, storing both the transform name and the poem for use by the Protocol. The Factory creates a single Deferred which represents the result of the transformation request. Notice how the Factory handles two error cases: a failure to connect and a connection that is closed before the poem is received. Also note the `clientConnectionLost` method is called even if we receive the poem, but in that case `self.deferred` will be `None`, thanks to the `handlePoem` method.

This Factory class creates the Deferred that it also fires. That's a good rule to follow in Twisted programming, so let's highlight it:

> In general, an object that makes a Deferred should also be in charge of firing that Deferred.

This "you make it, you fire it" rule helps ensure a given deferred is only fired once and makes it easier to follow the flow of control in a Twisted program.

In addition to the transform Factory, there is also a Proxy class which hides the details of making the TCP connection to a particular transform server:

```
 1  class TransformProxy(object):
 2      """
 3      I proxy requests to a transformation service.
 4      """
 5
 6      def __init__(self, host, port):
 7          self.host = host
 8          self.port = port
 9
10      def xform(self, xform_name, poem):
11          factory = TransformClientFactory(xform_name, poem)
12          from twisted.internet import reactor
```

```
13          reactor.connectTCP(self.host, self.port, factory)
14          return factory.deferred
```

This class presents a single `xform()` interface that other code can use to request transformations. So that other code can just request a transform and get a deferred back without mucking around with hostnames and port numbers.

The rest of the program is unchanged except for the `try_to_cummingsify` callback:

```
1  def try_to_cummingsify(poem):
2      d = proxy.xform('cummingsify', poem)
3
4      def fail(err):
5          print >>sys.stderr, 'Cummingsify failed!'
6          return poem
7
8      return d.addErrback(fail)
```

This callback now returns a deferred, but we didn't have to change the rest of the `main` function at all, other than to create the Proxy instance. Since `try_to_cummingsify` was part of a deferred chain (the deferred returned by `get_poetry`), it was already being used asynchronously and nothing else need change.

You'll note we are returning the result of `d.addErrback(fail)`. That's just a little bit of syntactic sugar. The `addCallback` and `addErrback` methods return the original deferred. We might just as well have written:

```
1  d.addErrback(fail)
2  return d
```

The first version is the same thing, just shorter.

Testing out the Client

The new client has a slightly different syntax than the others. If you have a transformation service running on port 10001 and two poetry servers running on ports 10002 and 10003, you would run:

```
python twisted-client-6/get-poetry.py 10001 10002 10003
```

To download two poems and transform them both. You can start the transform server like this:

```
python twisted-server-1/transformedpoetry.py --port 10001
```

And the poetry servers like this:

```
python twisted-server-1/fastpoetry.py --port 10002 poetry/fascination.txt
python twisted-server-1/fastpoetry.py --port 10003 poetry/science.txt
```

Then you can run the poetry client as above. After that, try crashing the transform server and re-running the client with the same command.

Wrapping Up

In this Part we learned how deferreds can transparently handle other deferreds in a callback chain, and thus we can safely add asynchronous callbacks to an 'outer' deferred without worrying about the details. That's pretty handy since lots of our functions are going to end up being asynchronous.

Do we know everything there is to know about deferreds yet? Not quite! There's one more important feature to talk about, but we'll save it for Part 14.

Suggested Exercises

1. Modify the client so we can ask for a specific kind of transformation by name.
2. Modify the client so the transformation server address is an optional argument. If it's not provided, skip the transformation step.
3. The `PoetryClientFactory` currently violates the "you make it, you fire it" rule for deferreds. Refactor `get_poetry` and `PoetryClientFactory` to remedy that.
4. Although we didn't demonstrate it, the case where an errback returns a deferred is symmetrical. Modify the `twisted-deferred/defer-10.py` example to verify it.
5. Find the place in the Deferred implementation that handles the case where a callback/errback returns another Deferred.

Part 14: When a Deferred Isn't

This continues the introduction started here. You can find an index to the entire series here.

Introduction

In this part we're going to learn another aspect of the `Deferred` class. To motivate the discussion, we'll add one more server to our stable of poetry-related services. Suppose we have a large number of internal clients who want to get poetry from the same external server. But this external server is slow and already over-burdened by the insatiable demand for poetry across the Internet. We don't want to contribute to that poor server's problems by sending all our clients there too.

So instead we'll make a caching proxy server. When a client connects to the proxy, the proxy will either fetch the poem from the external server or return a cached copy of a previously retrieved poem. Then we can point all our clients at the proxy and our contribution to the external server's load will be negligible. We illustrate this setup in Figure 30:

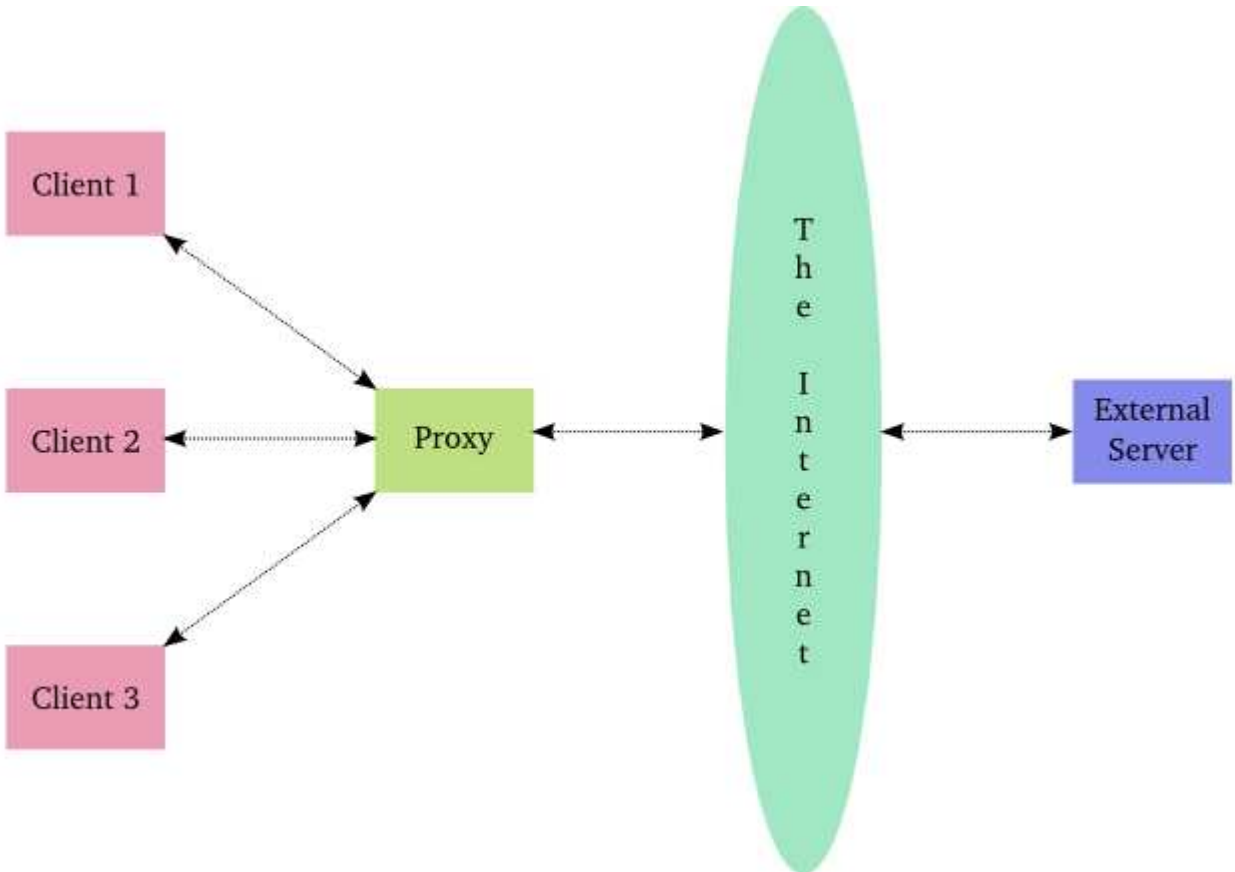

Figure 30: a caching proxy server

Consider what happens when a client connects to the proxy to get a poem. If the proxy's cache is empty, the proxy must wait (asynchronously) for the external server to respond before sending a poem back. So far so good, we already know how to handle that situation with an asynchronous function that returns a deferred. On the other hand, if there's already a poem in the cache, the proxy can send it back immediately, no need to wait at all. So the proxy's internal mechanism for getting a poem will sometimes be asynchronous and sometimes synchronous.

So what do we do if we have a function that is only asynchronous some of the time? Twisted provides a couple of options, and they both depend on a feature of the `Deferred` class we haven't used yet: you can fire a deferred *before* you return it to the caller.

This works because, although you cannot fire a deferred twice, you can add callbacks and errbacks to a deferred after it has fired. And when you do so, the deferred simply continues firing the chain from where it last left off. One important thing to note is an already-fired deferred may fire the new callback (or errback, depending on the state of the deferred) immediately, i.e., right when you add it.

Consider Figure 31, showing a deferred that has been fired:

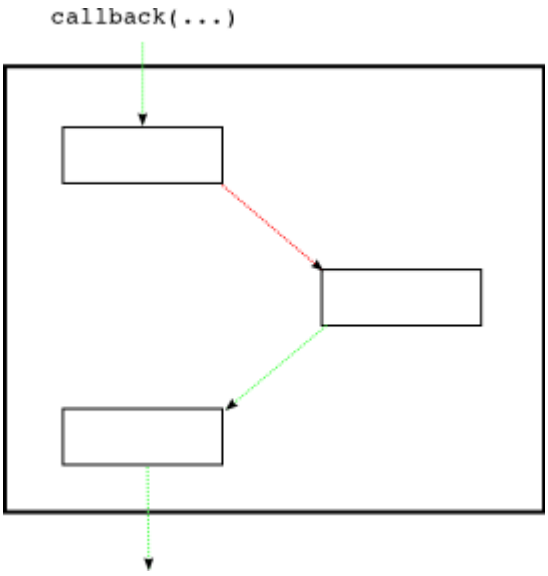

Figure 31: a deferred that has been fired

If we were to add another callback/errback pair at this point, then the deferred would immediately fire the new callback, as in Figure 32:

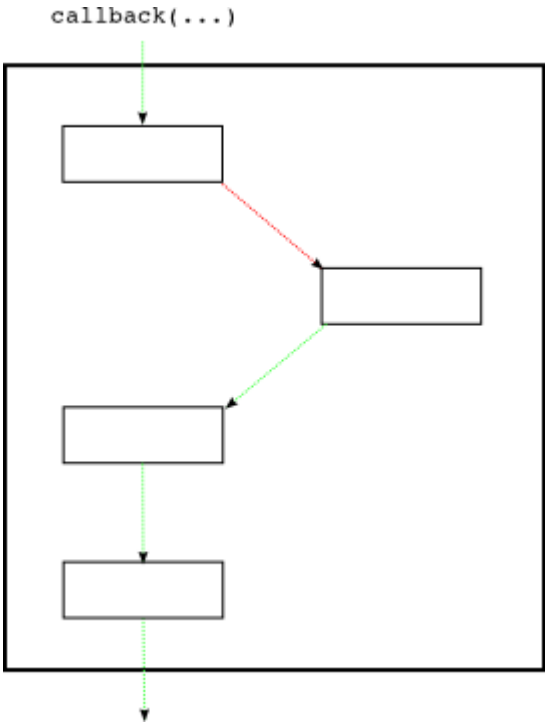

Figure 32: the same deferred with a new callback

The callback (not the errback) is fired because the previous callback succeeded. If it had failed (raised an Exception or returned a Failure) then the new errback would have been called instead.

We can test out this new feature with the example code in `twisted-deferred/defer-11.py`. Read and run that script to see how a deferred behaves when you fire it and then add callbacks. Note how in the first example each new callback is invoked immediately (you can tell from the order of the `print` output).

The second example in that script shows how we can `pause()` a deferred so it doesn't fire the callbacks right away. When we are ready for the callbacks to fire, we call `unpause()`. That's actually the same mechanism the deferred uses to pause itself when one of its callbacks returns another deferred. Nifty!

Proxy 1.0

Now let's look at the first version of the poetry proxy in `twisted-server-1/poetry-proxy.py`. Since the proxy acts as both a client and a server, it has two pairs of Protocol/Factory classes, one for serving up poetry, and one for getting a poem from the external server. We won't bother looking at the code for the client pair, it's the same as in previous poetry clients.

But before we look at the server pair, we'll look at the `ProxyService`, which the server-side protocol uses to get a poem:

```
 1  class ProxyService(object):
 2
 3      poem = None # the cached poem
 4
 5      def __init__(self, host, port):
 6          self.host = host
 7          self.port = port
 8
 9      def get_poem(self):
10          if self.poem is not None:
11              print 'Using cached poem.'
12              return self.poem
13
```

```
14          print 'Fetching poem from server.'
15          factory = PoetryClientFactory()
16          factory.deferred.addCallback(self.set_poem)
17          from twisted.internet import reactor
18          reactor.connectTCP(self.host, self.port, factory)
19          return factory.deferred
20
21      def set_poem(self, poem):
22          self.poem = poem
23          return poem
```

The key method there is `get_poem`. If there's already a poem in the cache, that method just returns the poem itself. On the other hand, if we haven't got a poem yet, we initiate a connection to the external server and return a deferred that will fire when the poem comes back. So `get_poem` is a function that is only asynchronous some of the time.

How do you handle a function like that? Let's look at the server-side protocol/factory pair:

```
1   class PoetryProxyProtocol(Protocol):
2
3       def connectionMade(self):
4           d = maybeDeferred(self.factory.service.get_poem)
5           d.addCallback(self.transport.write)
6           d.addBoth(lambda r: self.transport.loseConnection())
7
8   class PoetryProxyFactory(ServerFactory):
9
10      protocol = PoetryProxyProtocol
11
12      def __init__(self, service):
13          self.service = service
```

The factory is straightforward — it's just saving a reference to the proxy service so that protocol instances can call the `get_poem` method. The protocol is where the action is. Instead of calling `get_poem` directly, the protocol uses a wrapper function from the `twisted.internet.defer` module named `maybeDeferred`.

The `maybeDeferred` function takes a reference to another function, plus some optional arguments to call that function with (we aren't using any here). Then `maybeDeferred` will actually call that function and:

- If the function returns a deferred, `maybeDeferred` returns that same deferred, or
- If the function returns a Failure, `maybeDeferred` returns a new deferred that has been fired (via `.errback`) with that Failure, or
- If the function returns a regular value, `maybeDeferred` returns a deferred that has already been fired with that value as the result, or
- If the function raises an exception, `maybeDeferred` returns a deferred that has already been fired (via `.errback()`) with that exception wrapped in a Failure.

In other words, the return value from `maybeDeferred` is guaranteed to be a deferred, even if the function you pass in never returns a deferred at all. This allows us to safely call a synchronous function (even one that fails with an exception) and treat it like an asynchronous function returning a deferred.

> Note 1: There will still be a subtle difference, though. A deferred returned by a synchronous function has already been fired, so any callbacks or errbacks you add will run immediately, rather than in some future iteration of the reactor loop.
>
> Note 2: In hindsight, perhaps naming a function that always returns a deferred "maybeDeferred" was not the best choice, but there you go.

Once the protocol has a real deferred in hand, it can just add some callbacks that send the poem to the

client and then close the connection. And that's it for our first poetry proxy!

Running the Proxy

To try out the proxy, start up a poetry server, like this:

```
python twisted-server-1/fastpoetry.py --port 10001 poetry/fascination.txt
```

And now start a proxy server like this:

```
python twisted-server-1/poetry-proxy.py --port 10000 10001
```

It should tell you that it's proxying poetry on port 10000 for the server on port 10001.
Now you can point a client at the proxy:

```
python twisted-client-4/get-poetry.py 10000
```

We'll use an earlier version of the client that isn't concerned with poetry transformations. You should see the poem appear in the client window and some text in the proxy window saying it's fetching the poem from the server. Now run the client again and the proxy should confirm it is using the cached version of the poem, while the client should show the same poem as before.

Proxy 2.0

As we mentioned earlier, there's an alternative way to implement this scheme. This is illustrated in Poetry Proxy 2.0, located in `twisted-server-2/poetry-proxy.py`. Since we can fire deferreds before we return them, we can make the proxy service return an already-fired deferred when there's already a poem in the cache. Here's the new version of the `get_poem` method on the proxy service:

```
1  def get_poem(self):
2      if self.poem is not None:
3          print 'Using cached poem.'
4          # return an already-fired deferred
5          return succeed(self.poem)
6
7      print 'Fetching poem from server.'
8      factory = PoetryClientFactory()
9      factory.deferred.addCallback(self.set_poem)
10     from twisted.internet import reactor
11     reactor.connectTCP(self.host, self.port, factory)
12     return factory.deferred
```

The `defer.succeed` function is just a handy way to make an already-fired deferred given a result. Read the implementation for that function and you'll see it's simply a matter of making a new deferred and then firing it with `.callback()`. If we wanted to return an already-failed deferred we could use `defer.fail` instead.

In this version, since `get_poem` always returns a deferred, the protocol class no longer needs to use `maybeDeferred` (though it would still work if it did, as we learned above):

```
1  class PoetryProxyProtocol(Protocol):
2
3      def connectionMade(self):
4          d = self.factory.service.get_poem()
5          d.addCallback(self.transport.write)
6          d.addBoth(lambda r: self.transport.loseConnection())
```

Other than these two changes, the second version of the proxy is just like the first, and you can run it in the same way we ran the original version.

Summary

In this Part we learned how deferreds can be fired before they are returned, and thus we can use them in synchronous (or sometimes synchronous) code. And we have two ways to do that:

- We can use `maybeDeferred` to handle a function that sometimes returns a deferred and other times returns a regular value (or throws an exception), or
- We can pre-fire our own deferreds, using `defer.succeed` and `defer.fail`, so our "semi-synchronous" functions always return a deferred no matter what.

Which technique we choose is really up to us. The former emphasizes the fact that our functions aren't always asynchronous while the latter makes the client code simpler. Perhaps there's not a definitive argument for choosing one over the other.

Both techniques are made possible because we can add callbacks and errbacks to a deferred after it has fired. And that explains the curious fact we discovered in Part 9 and the `twisted-deferred/defer-unhandled.py` example. We learned that an "unhandled error" in a deferred, in which either the last callback or errback fails, isn't reported until the deferred is garbage collected (i.e., there are no more references to it in user code). Now we know why — since we could always add another callback pair to a deferred which does handle that error, it's not until the last reference to a deferred is dropped that Twisted can say the error was not handled.

> Now that you've spent so much time exploring the `Deferred` class, which is located in the `twisted.internet` package, you may have noticed it doesn't actually have anything to do with the Internet. It's just an abstraction for managing callbacks. So what's it doing there? That is an artifact of Twisted's history. In the best of all possible worlds (where I am paid millions of dollars to play in the World Ultimate Frisbee League), the `defer` module would probably be in `twisted.python`. Of course, in that world you would probably be too busy fighting crime with your super-powers to read this introduction. I suppose that's life.

So is that it for deferreds? Do we finally know all their features? For the most part, we do. But Twisted includes alternate ways of using deferreds that we haven't explored yet (we'll get there!). And in the meantime, the Twisted developers have been beavering away adding new stuff. In an upcoming release, the `Deferred` class will acquire a brand new capability. We'll introduce it in a future Part, but first we'll take a break from deferreds and look at some other aspects of Twisted, including testing in Part 15.

Suggested Exercises

1. Modify the `twisted-deferred/defer-11.py` example to illustrate pre-failing deferreds using `.errback()`. Read the documentation and implementation of the `defer.fail` function.
2. Modify the proxy so that a cached poem older than 2 hours is discarded, causing the next poetry request to re-request it from the server
3. The proxy is supposed to avoid contacting the server more than once, but if several client requests come in at the same time when there is no poem in the cache, the proxy will make multiple poetry requests. It's easier to see if you use a slow server to test it out.

   Modify the proxy service so that only one request is generated. Right now the service only has two states: either the poem is in the cache or it isn't. You will need to recognize a third state indicating a request has been made but not completed. When the `get_poem` method is called in the third state, add a new deferred to a list of 'waiters'. That new deferred will be the result of the `get_poem` method. When the poem finally comes back, fire all the waiting deferreds with the poem and transition to the cached state. On the other hand, if the poem fails, fire the `.errback()` method of all the waiters and transition to the non-cached state.

4. Add a transformation proxy to the proxy service. This service should work like the original

transformation service, but use an external server to do the transformations.

5. Consider this hypothetical piece of code:

```
1  d = some_async_function() # d is a Deferred
2  d.addCallback(my_callback)
3  d.addCallback(my_other_callback)
4  d.addErrback(my_errback)
```

Suppose that when the deferred `d` is returned on line 1, it has not been fired. Is it possible for that deferred
to fire while we are adding our callbacks and errback on lines 2-4? Why or why not?

Part 15: Tested Poetry

This continues the introduction started here. You can find an index to the entire series here.

Introduction

We've written a lot of code in our exploration of Twisted, but so far we've neglected to write something important — tests. And you may be wondering how you can test asynchronous code using a synchronous framework like the `unittest` package that comes with Python. The short answer is you can't. As we've discovered, synchronous and asynchronous code do not mix, at least not readily.

Fortunately, Twisted includes its own testing framework called `trial` that does support testing asynchronous code (and you can use it to test synchronous code, too).

We'll assume you are already familiar with the basic mechanics of `unittest` and similar testing frameworks, in which you create tests by defining a class with a specific parent class (usually called something like `TestCase`), and each method of that class starting with the word "`test`" is considered a single test. The framework takes care of discovering all the tests, running them one after the other with optional `setUp` and `tearDown` steps, and then reporting the results.

The Example

You will find some example tests located in `tests/test_poetry.py`. To ensure all our examples are self-contained (so you don't need to worry about `PYTHONPATH` settings), we have copied all the necessary code into the test module. Normally, of course, you would just import the modules you wanted to test.

The example is testing both the poetry client and server, by using the client to fetch a poem from a test server. To provide a poetry server for testing, we implement the `setUp` method in our test case:

```
1  class PoetryTestCase(TestCase):
2
3      def setUp(self):
4          factory = PoetryServerFactory(TEST_POEM)
5          from twisted.internet import reactor
6          self.port = reactor.listenTCP(0, factory, interface="127.0.0.1")
7          self.portnum = self.port.getHost().port
```

The `setUp` method makes a poetry server with a test poem, and listens on a random, open port. We save the port number so the actual tests can use it, if they need to. And, of course, we clean up the test server in `tearDown` when the test is done:

```
1  def tearDown(self):
2      port, self.port = self.port, None
3      return port.stopListening()
```

That brings us to our first test, `test_client`, where we use `get_poetry` to retrieve the poem from the test server and verify it's the poem we expected:

```
def test_client(self):
    """The correct poem is returned by get_poetry."""
    d = get_poetry('127.0.0.1', self.portnum)

    def got_poem(poem):
        self.assertEquals(poem, TEST_POEM)

    d.addCallback(got_poem)

    return d
```

Notice that our test function is returning a deferred. Under `trial`, each test method runs as a callback. That means the reactor is running and we can perform asynchronous operations as part of the test. We just need to let the framework know that our test is asynchronous and we do that in the usual Twisted way — return a deferred.

The `trial` framework will wait until the deferred fires before calling the `tearDown` method, and will fail the test if the deferred fails (i.e., if the last callback/errback pair fails). It will also fail the test if our deferred takes too long to fire, two minutes by default. And that means if the test finished, we know our deferred fired, and therefore our callback fired and ran the `assertEquals` test method.

Our second test, `test_failure`, verifies that `get_poetry` fails in the appropriate way if we can't connect to the server:

```
def test_failure(self):
    """The correct failure is returned by get_poetry when
    connecting to a port with no server."""
    d = get_poetry('127.0.0.1', -1)
    return self.assertFailure(d, ConnectionRefusedError)
```

Here we attempt to connect to an invalid port and then use the `trial`-provided `assertFailure` method. This method is like the familiar `assertRaises` method but for asynchronous code. It returns a deferred that succeeds if the given deferred fails with the given exception, and fails otherwise.

You can run the tests yourself using the `trial` script like this:

```
trial tests/test_poetry.py
```

And you should see some output showing each test case and an `OK` telling you each test passed.

Discussion

Because `trial` is so similar to `unittest` when it comes to the basic API, it's pretty easy to get started writing tests. Just return a deferred if your test uses asynchronous code, and `trial` will take care of the rest. You can also return a deferred from the `setUp` and `tearDown` methods, if those need to be asynchronous as well.

Any log messages from your tests will be collected in a file inside a directory called `_trial_temp` that `trial` will create automatically if it doesn't exist. In addition to the errors printed to the screen, the log is a useful starting point when debugging failing tests.

Figure 33 shows a hypothetical test run in progress:

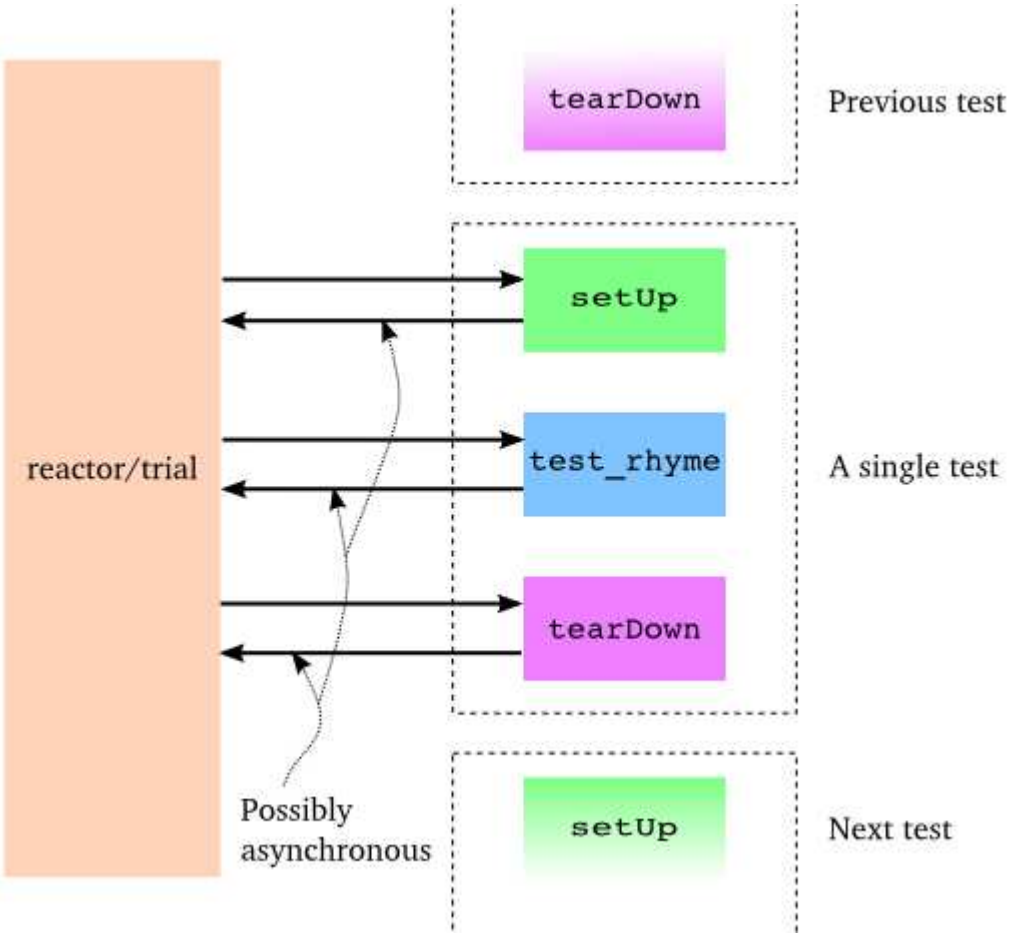

Figure 33: a trial test in progress

If you've used similar frameworks before, this should be a familiar model, except that all the test-related methods may return deferreds.

The `trial` framework is also a good illustration of how "going asynchronous" involves changes that cascade throughout the program. In order for a test (or any function or method) to be asynchronous, it must:

1. Not block and, usually,
2. return a deferred.

But that means that whatever calls that function must be willing to accept a deferred, and also not block (and thus likely return a deferred as well). And so it goes up and up. Thus, the need for a framework like `trial` which can handle asynchronous tests that return deferreds.

Summary

That's it for our look at unit testing. If would like to see more examples of how to write unit tests for Twisted code, you need look no further than Twisted itself. The Twisted framework comes with a very large suite of unit tests, with new ones added in each release. Since these tests are scrutinized by Twisted experts during code reviews before being accepted into the codebase, they make excellent examples of how to test Twisted code the right way.

In Part 16 we will use a Twisted utility to turn our poetry server into a genuine daemon.

Suggested Exercises

1. Change one of the tests to make it fail and run `trial` again to see the output.

2. Read the online trial documentation.
3. Write tests for some of the other poetry services we have created in this series.
4. Explore some of the tests in Twisted.

Part 16: Twisted Daemonologie

This continues the introduction started here. You can find an index to the entire series here.

Introduction

The servers we have written so far have just run in a terminal window, with output going to the screen via `print` statements. This works alright for development, but it's hardly a way to deploy services in production. A well-behaved production server ought to:

1. Run as a daemon process, unconnected with any terminal or user session. You don't want a service to shut down just because the administrator logs out.
2. Send debugging and error output to a set of rotated log files, or to the `syslog` service.
3. Drop excessive privileges, e.g., switching to a lower-privileged user before running.
4. Record its `pid` in a file so that the administrator can easily send signals to the daemon.

We can get all of those features using the `twistd` script provided by Twisted. But first we'll have to change our code a bit.

The Concepts

Understanding `twistd` will require learning a few new concepts in Twisted, the most important being a `Service`. As usual, several of the new concepts are accompanied by new `Interface`s.

IService

The `IService` interface defines a named service that can be started and stopped. What does the service do? Whatever you like — rather than define the specific function of the service, the interface requires only that it provide a small set of generic attributes and methods.

There are two required attributes: `name` and `running`. The `name` attribute is just a string, like `'fastpoetry'`, or `None` if you don't want to give your service a name. The `running` attribute is a Boolean value and is true if the service has been successfully started.

We're only going to touch on some of the methods of `IService`. We'll skip some that are obvious, and others that are more advanced and often go unused in simpler Twisted programs. The two principle methods of `IService` are `startService` and `stopService`:

```
1  def startService():
2      """
3      Start the service.
4      """
5
6  def stopService():
7      """
8      Stop the service.
9
10     @rtype: L{Deferred}
11     @return: a L{Deferred} which is triggered when the service has
12         finished shutting down. If shutting down is immediate, a
13         value can be returned (usually, C{None}).
14     """
```

Again, what these methods actually do will depend on the service in question. For example, the `startService` method might:

- Load some configuration data, or
- Initialize a database, or
- Start listening on a port, or
- Do nothing at all.

And the `stopService` method might:

- Persist some state, or
- Close open database connections, or
- Stop listening on a port, or
- Do nothing at all.

When we write our own custom services we'll need to implement these methods appropriately. For some common behaviors, like listening on a port, Twisted provides ready-made services we can use instead.

Notice that `stopService` may optionally return a deferred, which is required to fire when the service has completely shut down. This allows our services to finish cleaning up after themselves before the entire application terminates. If your service shuts down immediately you can just return `None` instead of a deferred.

Services can be organized into collections that get started and stopped together. The last `IService` method we're going to look at, `setServiceParent`, adds a Service to a collection:

```
1  def setServiceParent(parent):
2      """
3      Set the parent of the service.
4
5      @type parent: L{IServiceCollection}
6      @raise RuntimeError: Raised if the service already has a parent
7          or if the service has a name and the parent already has a child
8          by that name.
9      """
```

Any service can have a parent, which means services can be organized in a hierarchy. And that brings us to the next `Interface` we're going to look at today.

IServiceCollection

The `IServiceCollection` interface defines an object which can contain `IService` objects. A service collection is a just plain container class with methods to:

- Look up a service by name (`getServiceNamed`)
- Iterate over the services in the collection (`__iter__`)
- Add a service to the collection (`addService`)
- Remove a service from the collection (`removeService`)

Note that an implementation of `IServiceCollection` isn't automatically an implementation of `IService`, but there's no reason why one class can't implement both interfaces (and we'll see an example of that shortly).

Application

A Twisted `Application` is not defined by a separate interface. Rather, an `Application` object is required to implement both `IService` and `IServiceCollection`, as well as a few other interfaces we aren't going

to cover.

An `Application` is the top-level service that represents your entire Twisted application. All the other services in your daemon will be children (or grandchildren, etc.) of the `Application` object.

It is rare to actually implement your own `Application`. Twisted provides an implementation that we'll use today.

Twisted Logging

Twisted includes its own logging infrastructure in the module `twisted.python.log`. The basic API for writing to the log is simple, so we'll just include a short example located in `basic-twisted/log.py`, and you can skim the Twisted module for details if you are interested.

We won't bother showing the API for installing logging handlers, since `twistd` will do that for us.

FastPoetry 2.0

Alright, let's look at some code. We've updated the fast poetry server to run with `twistd`. The source is located in `twisted-server-3/fastpoetry.py`. First we have the poetry protocol:

```
1  class PoetryProtocol(Protocol):
2
3      def connectionMade(self):
4          poem = self.factory.service.poem
5          log.msg('sending %d bytes of poetry to %s'
6                  % (len(poem), self.transport.getPeer()))
7          self.transport.write(poem)
8          self.transport.loseConnection()
```

Notice instead of using a `print` statement, we're using the `twisted.python.log.msg` function to record each new connection.
Here's the factory class:

```
1  class PoetryFactory(ServerFactory):
2
3      protocol = PoetryProtocol
4
5      def __init__(self, service):
6          self.service = service
```

As you can see, the poem is no longer stored on the factory, but on a service object referenced by the factory. Notice how the protocol gets the poem from the service via the factory. Finally, here's the service class itself:

```
1  class PoetryService(service.Service):
2
3      def __init__(self, poetry_file):
4          self.poetry_file = poetry_file
5
6      def startService(self):
7          service.Service.startService(self)
8          self.poem = open(self.poetry_file).read()
9          log.msg('loaded a poem from: %s' % (self.poetry_file,))
```

As with many other `Interface` classes, Twisted provides a base class we can use to make our own implementations, with helpful default behaviors. Here we use the `twisted.application.service.Service` class to implement our `PoetryService`.

The base class provides default implementations of all required methods, so we only need to implement

the ones with custom behavior. In this case, we just override `startService` to load the poetry file. Note we still call the base class method (which sets the `running` attribute for us).

Another point is worth mentioning. The `PoetryService` object doesn't know anything about the details of the `PoetryProtocol`. The service's only job is to load the poem and provide access to it for any object that might need it. In other words, the `PoetryService` is entirely concerned with the higher-level details of providing poetry, rather than the lower-level details of sending a poem down a TCP connection. So this same service could be used by another protocol, say UDP or XML-RPC. While the benefit is rather small for our simple service, you can imagine the advantage for a more realistic service implementation.

If this were a typical Twisted program, all the code we've looked at so far wouldn't actually be in this file. Rather, it would be in some other module(s) (perhaps `fastpoetry.protocol` and `fastpoetry.service`). But following our usual practice of making these examples self-contained, we've including everything we need in a single script.

Twisted `tac` files

The rest of the script contains what would normally be the entire content — a Twisted `tac` file. A `tac` file is a Twisted Application Configuration file that tells `twistd` how to construct an application. As a configuration file it is responsible for choosing settings (like port numbers, poetry file locations, etc.) to run the application in some particular way. In other words, a `tac` file represents a specific deployment of our service (serve *that* poem on *this* port) rather than a general script for starting any poetry server.

If we were running multiple poetry servers on the same host, we would have a `tac` file for each one (so you can see why `tac` files normally don't contain any general-purpose code). In our example, the `tac` file is configured to serve `poetry/ecstasy.txt` run on port `10000` of the loopback interface:

```
1 # configuration parameters
2 port = 10000
3 iface = 'localhost'
4 poetry_file = 'poetry/ecstasy.txt'
```

Note that `twistd` doesn't know anything about these particular variables, we just define them here to keep all our configuration values in one place. In fact, `twistd` only really cares about one variable in the entire file, as we'll see shortly. Next we begin building up our application:

```
1 # this will hold the services that combine to form the poetry server
2 top_service = service.MultiService()
```

Our poetry server is going to consist of two services, the `PoetryService` we defined above, and a Twisted built-in service that creates the listening socket our poem will be served from. Since these two services are clearly related to each other, we'll group them together using a `MultiService`, a Twisted class which implements both `IService` and `IServiceCollection`.

As a service collection, the `MultiService` will group our two poetry services together. And as a service, the `MultiService` will start both child services when the `MultiService` itself is started, and stop both child services when it is stopped. Let's add the first poetry service to the collection:

```
1 # the poetry service holds the poem. it will load the poem when it is
2 # started
3 poetry_service = PoetryService(poetry_file)
4 poetry_service.setServiceParent(top_service)
```

This is pretty simple stuff. We just create the `PoetryService` and then add it to the collection with `setServiceParent`, a method we inherited from the Twisted base class. Next we add the TCP listener:

```
1 # the tcp service connects the factory to a listening socket. it will
2 # create the listening socket when it is started
```

```
3 factory = PoetryFactory(poetry_service)
4 tcp_service = internet.TCPServer(port, factory, interface=iface)
5 tcp_service.setServiceParent(top_service)
```

Twisted provides the `TCPServer` service for creating a TCP listening socket connected to an arbitrary factory (in this case our `PoetryFactory`). We don't call `reactor.listenTCP` directly because the job of a `tac` file is to get our application ready to start, without actually starting it. The `TCPServer` will create the socket after it is started by `twistd`.

You might have noticed we didn't bother to give any of our services names. Naming services is not required, but only an optional feature you can use if you want to 'look up' services at runtime. Since we don't need to do that in our little application, we don't bother with it here.

Ok, now we've got both our services combined into a collection. Now we just make our `Application` and add our collection to it:

```
1 # this variable has to be named 'application'
2 application = service.Application("fastpoetry")
3
4 # this hooks the collection we made to the application
5 top_service.setServiceParent(application)
```

The only variable in this script that `twistd` really cares about is the `application` variable. That is how `twistd` will find the application it's supposed to start (and so the variable has to be named 'application'). And when the application is started, all the services we added to it will be started as well.

Figure 34 shows the structure of the application we just built:

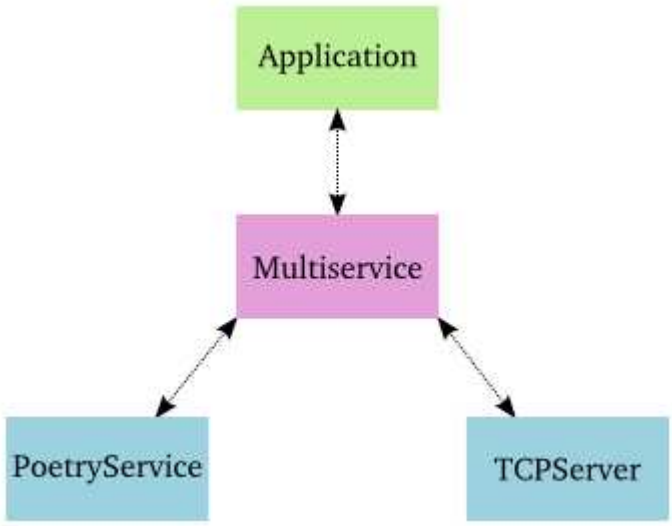

Figure 34: the structure of our fastpoetry application

Running the Server

Let's take our new server for a spin. As a `tac` file, we need to start it with `twistd`. Of course, it's also just a regular Python file, too. So let's run it with Python first and see what happens:

```
python twisted-server-3/fastpoetry.py
```

If you do this, you'll find that what happens is nothing! As we said before, the job of a `tac` file is to get an application ready to run, without actually running it. As a reminder of this special purpose of `tac` files, some people name them with a `.tac` extension instead of `.py`. But the `twistd` script doesn't actually care about the extension.

Let's run our server for real, using `twistd`:

```
twistd --nodaemon --python twisted-server-3/fastpoetry.py
```

After running that command, you should see some output like this:

```
2010-06-23 20:57:14-0700 [-] Log opened.
2010-06-23 20:57:14-0700 [-] twistd 10.0.0 (/usr/bin/python 2.6.5) starting up.
2010-06-23 20:57:14-0700 [-] reactor class: twisted.internet.selectreactor.SelectReact
2010-06-23 20:57:14-0700 [-] __builtin__.PoetryFactory starting on 10000
2010-06-23 20:57:14-0700 [-] Starting factory <__builtin__.PoetryFactory instance at 0
2010-06-23 20:57:14-0700 [-] loaded a poem from: poetry/ecstasy.txt
```

Here's a few things to notice:

1. You can see the output of the Twisted logging system, including the `PoetryFactory`'s call to `log.msg`. But we didn't install a logger in our `tac` file, so `twistd` must have installed one for us.
2. You can also see our two main services, the `PoetryService` and the `TCPServer` starting up.
3. The shell prompt never came back. That means our server isn't running as a daemon. By default, `twistd` does run a server as a daemon process (that's the main reason `twistd` exists), but if you include the `--nodaemon` option then `twistd` will run your server as a regular shell process instead, and will direct the log output to standard output as well. This is useful for debugging your `tac` files.

Now test out the server by fetching a poem, either with one of our poetry clients or just `netcat`:

```
netcat localhost 10000
```

That should fetch the poem from the server and you should see a new log line like this:

```
2010-06-27 22:17:39-0700 [__builtin__.PoetryFactory] sending 3003 bytes of poetry to I
```

That's from the call to `log.msg` in `PoetryProtocol.connectionMade`. As you make more requests to the server, you will see additional log entries for each request.

Now stop the server by pressing `Ctrl-C`. You should see some output like this:

```
^C2010-06-29 21:32:59-0700 [-] Received SIGINT, shutting down.
2010-06-29 21:32:59-0700 [-] (Port 10000 Closed)
2010-06-29 21:32:59-0700 [-] Stopping factory <__builtin__.PoetryFactory instance at 0
2010-06-29 21:32:59-0700 [-] Main loop terminated.
2010-06-29 21:32:59-0700 [-] Server Shut Down.
```

As you can see, Twisted does not simply crash, but shuts itself down cleanly and tells you about it with log messages. Notice our two main services shutting themselves down as well.

Ok, now start the server up once more:

```
twistd --nodaemon --python twisted-server-3/fastpoetry.py
```

Then open another shell and change to the `twisted-intro` directory. A directory listing should show a file called `twistd.pid`. This file is created by `twistd` and contains the process ID of our running server. Try executing this alternative command to shut down the server:

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

Notice that `twistd` cleans up the process ID file when our server shuts down.

A Real Daemon

Now let's start our server as an actual daemon process, which is even simpler to do as it's `twistd`'s

default behavior:

```
twistd --python twisted-server-3/fastpoetry.py
```

This time we get our shell prompt back almost immediately. And if you list the contents of your directory you will see, in addition to the `twistd.pid` file for the server we just ran, a `twistd.log` file with the log entries that were formerly displayed at the shell prompt.

When starting a daemon process, `twistd` installs a log handler that writes entries to a file instead of standard output. The default log file is `twistd.log`, located in the same directory where you ran `twistd`, but you can change that with the `--logfile` option if you wish. The handler that `twistd` installs also rotates the log whenever the size exceeds one megabyte.

You should be able to see the server running by listing all the processes on your system. Go ahead and test out the server by fetching another poem. You should see new entries appear in the log file for each poem you request.

Since the server is no longer connected to the shell (or any other process except `init`), you cannot shut it down with `Ctrl-C`. As a true daemon process, it will continue to run even if you log out. But we can still use the `twistd.pid` file to stop the process:

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

And when that happens the shutdown messages appear in the log, the `twistd.pid` file is removed, and our server stops running. Neato.

It's a good idea to check out some of the other `twistd` startup options. For example, you can tell `twistd` to switch to a different user or group account before starting the daemon (typically a way to drop privileges your server doesn't need as a security precaution). We won't bother going into those extra options, you can find them using the `--help` switch to `twistd`.

The Twisted Plugin System

Ok, now we can use `twistd` to start up our servers as genuine daemon processes. This is all very nice, and the fact that our "configuration" files are really just Python source files gives us a great deal of flexibility in how we set things up. But we don't always need that much flexibility. For our poetry servers, we typically only have a few options we might care about:

1. The poem to serve.
2. The port to serve it from.
3. The interface to listen on.

Making new `tac` files for simple variations on those values seems rather excessive. It would be nice if we could just specify those values as options on the `twistd` command line. The Twisted plugin system allows us to do just that.

Twisted plugins provide a way of defining named Applications, with a custom set of command-line options, that `twistd` can dynamically discover and run. Twisted itself comes with a set of built-in plugins. You can see them all by running `twistd` without any arguments. Try running it now, but outside of the `twisted-intro` directory. After the help section, you should see some output like this:

```
...
ftp                 An FTP server.
telnet              A simple, telnet-based remote debugging service.
socks               A SOCKSv4 proxy service.
...
```

Each line shows one of the built-in plugins that come with Twisted. And you can run any of them using

`twistd`.
Each plugin also comes with its own set of options, which you can discover using `--help`. Let's see what the options for the `ftp` plugin are:

```
twistd ftp --help
```

Note that you need to put the `--help` switch after the `ftp` command, since you want the options for the `ftp` plugin rather than for `twistd` itself.
We can run the `ftp` server with `twistd` just like we ran our poetry server. But since it's a plugin, we just run it by name:

```
twistd --nodaemon ftp --port 10001
```

That command runs the `ftp` plugin in non-daemon mode on port 10001. Note the `twistd` option `nodaemon` comes before the plugin name, while the plugin-specific option `port` comes after the plugin name. As with our poetry server, you can stop that plugin with `Ctrl-C`.

Ok, let's turn our poetry server into a Twisted plugin. First we need to introduce a couple of new concepts.

IPlugin

Any Twisted plugin must implement the `twisted.plugin.IPlugin` interface. If you look at the declaration of that `Interface`, you'll find it doesn't actually specify any methods. Implementing `IPlugin` is simply a way for a plugin to say "Hello, I'm a plugin!" so `twistd` can find it. Of course, to be of any use, it will have to implement some other interface and we'll get to that shortly.

But how do you know if an object actually implements an empty interface? The `zope.interface` package includes a function called `implements` that you can use to declare that a particular class implements a particular interface. We'll see an example of that in the plugin version of our poetry server.

IServiceMaker

In addition to `IPlugin`, our plugin will implement the `IServiceMaker` interface. An object which implements `IServiceMaker` knows how to create an `IService` that will form the heart of a running application. `IServiceMaker` specifies three attributes and a method:

1. `tapname`: a string name for our plugin. The "tap" stands for Twisted Application Plugin. Note: an older version of Twisted also made use of pickled application files called "tapfiles", but that functionality is deprecated.
2. `description`: a description of the plugin, which `twistd` will display as part of its help text.
3. `options`: an object which describes the command-line options this plugin accepts.
4. `makeService`: a method which creates a new `IService` object, given a specific set of command-line options

We'll see how all this gets put together in the next version of our poetry server.

Fast Poetry 3.0

Now we're ready to take a look at the plugin version of Fast Poetry, located in `twisted/plugins/fastpoetry_plugin.py`.

You might notice we've named these directories differently than any of the other examples. That's because `twistd` requires plugin files to be located in a `twisted/plugins` directory, located in your Python module search path. The directory doesn't have to be a package (i.e., you don't need any `__init__.py` files) and you can have multiple `twisted/plugins` directories on your path and `twistd`

will find them all. The actual filename you use for the plugin doesn't matter either, but it's still a good idea to name it according to the application it represents, like we have done here.

The first part of our plugin contains the same poetry protocol, factory, and service implementations as our `tac` file. And as before, this code would normally be in a separate module but we've placed it in the plugin to make the example self-contained.

Next comes the declaration of the plugin's command-line options:

```
class Options(usage.Options):

    optParameters = [
        ['port', 'p', 10000, 'The port number to listen on.'],
        ['poem', None, None, 'The file containing the poem.'],
        ['iface', None, 'localhost', 'The interface to listen on.'],
        ]
```

This code specifies the plugin-specific options that a user can place after the plugin name on the `twistd` command line. We won't go into details here as it should be fairly clear what is going on. Now we get to the main part of our plugin, the service maker class:

```
class PoetryServiceMaker(object):

    implements(service.IServiceMaker, IPlugin)

    tapname = "fastpoetry"
    description = "A fast poetry service."
    options = Options

    def makeService(self, options):
        top_service = service.MultiService()

        poetry_service = PoetryService(options['poem'])
        poetry_service.setServiceParent(top_service)

        factory = PoetryFactory(poetry_service)
        tcp_service = internet.TCPServer(int(options['port']), factory,
                                         interface=options['iface'])
        tcp_service.setServiceParent(top_service)

        return top_service
```

Here you can see how the `zope.interface.implements` function is used to declare that our class implements both `IServiceMaker` and `IPlugin`.

You should recognize the code in `makeService` from our earlier `tac` file implementation. But this time we don't need to make an `Application` object ourselves, we just create and return the top level service that our application will run and `twistd` will take care of the rest. Notice how we use the `options` argument to retrieve the plugin-specific command-line options given to `twistd`.

After declaring that class, there's only on thing left to do:

```
service_maker = PoetryServiceMaker()
```

The `twistd` script will discover that instance of our plugin and use it to construct the top level service. Unlike the `tac` file, the variable name we choose is irrelevant. What matters is that our object implements both `IPlugin` and `IServiceMaker`.

Now that we've created our plugin, let's run it. Make sure that you are in the `twisted-intro` directory, or that the `twisted-intro` directory is in your python module search path. Then try running `twistd` by itself. You should now see that "fastpoetry" is one of the plugins listed, along with the description text

from our plugin file.

You will also notice that a new file called `dropin.cache` has appeared in the `twisted/plugins` directory. This file is created by `twistd` to speed up subsequent scans for plugins.

Now let's get some help on using our plugin:

```
twistd fastpoetry --help
```

You should see the options that are specific to the fastpoetry plugin in the help text. Finally, let's run our plugin:

```
twistd fastpoetry --port 10000 --poem poetry/ecstasy.txt
```

That will start a fastpoetry server running as a daemon. As before, you should see both `twistd.pid` and `twistd.log` files in the current directory. After testing out the server, you can shut it down:

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

And that's how you make a Twisted plugin.

Summary

In this Part we learned about turning our Twisted servers into long-running daemons. We touched on the Twisted logging system and on how to use `twistd` to start a Twisted application as a daemon process, either from a `tac` configuration file or a Twisted plugin. In Part 17 we'll return to the more fundamental topic of asynchronous programming and look at another way of structuring our callbacks in Twisted.

Suggested Exercises

1. Modify the `tac` file to serve a second poem on another port. Keep the services for each poem separate by using another `MultiService` object.
2. Create a new `tac` file that starts a poetry proxy server.
3. Modify the plugin file to accept an optional second poetry file and second port to serve it on.
4. Create a new plugin for the poetry proxy server.

Part 17: Just Another Way to Spell "Callback"

This continues the introduction started here. You can find an index to the entire series here.

Introduction

In this Part we're going to return to the subject of callbacks. We'll introduce another technique for writing callbacks in Twisted that uses generators. We'll show how the technique works and contrast it with using "pure" Deferreds. Finally we'll rewrite one of our poetry clients using this technique. But first let's review how generators work so we can see why they are a candidate for creating callbacks.

A Brief Review of Generators

As you probably know, a Python generator is a "restartable function" that you create by using the `yield` expression in the body of your function. By doing so, the function becomes a "generator function" that returns an iterator you can use to run the function in a series of steps. Each cycle of the iterator restarts the function, which proceeds to execute until it reaches the next `yield`.

Generators (and iterators) are often used to represent lazily-created sequences of values. Take a look at

the example code in inline-callbacks/gen-1.py:

```
1  def my_generator():
2      print 'starting up'
3      yield 1
4      print "workin'"
5      yield 2
6      print "still workin'"
7      yield 3
8      print 'done'
9
10 for n in my_generator():
11     print n
```

Here we have a generator that creates the sequence 1, 2, 3. If you run the code, you will see the `print` statements in the generator interleaved with the `print` statement in the `for` loop as the loop cycles through the generator.

We can make this code more explicit by creating the generator ourselves (inline-callbacks/gen-2.py):

```
1  def my_generator():
2      print 'starting up'
3      yield 1
4      print "workin'"
5      yield 2
6      print "still workin'"
7      yield 3
8      print 'done'
9
10 gen = my_generator()
11
12 while True:
13     try:
14         n = gen.next()
15     except StopIteration:
16         break
17     else:
18         print n
```

Considered as a sequence, the generator is just an object for getting successive values. But we can also view things from the point of view of the generator itself:

1. The generator function doesn't start running until "called" by the loop (using the `next` method).
2. Once the generator is running, it keeps running until it "returns" to the loop (using `yield`).
3. When the loop is running other code (like the `print` statement), the generator is not running.
4. When the generator is running, the loop is not running (it's "blocked" waiting for the generator).
5. Once a generator `yield`s control to the loop, an arbitrary amount of time may pass (and an arbitrary amount of other code may execute) until the generator runs again.

This is very much like the way callbacks work in an asynchronous system. We can think of the `while` loop as the reactor, and the generator as a series of callbacks separated by `yield` statements, with the interesting fact that all the callbacks share the same local variable namespace, and the namespace persists from one callback to the next.

Furthermore, you can have multiple generators active at once (see the example in inline-callbacks/gen-3.py), with their "callbacks" interleaved with each other, just as you can have independent asynchronous tasks running in a system like Twisted.

Something is still missing, though. Callbacks aren't just called by the reactor, they also receive information. When part of a deferred's chain, a callback either receives a result, in the form of a single

Python value, or an error, in the form of a `Failure`.

Starting with Python 2.5, generators were extended in a way that allows you to send information to a generator when you restart it, as illustrated in [inline-callbacks/gen-4.py]:

```
class Malfunction(Exception):
    pass

def my_generator():
    print 'starting up'

    val = yield 1
    print 'got:', val

    val = yield 2
    print 'got:', val

    try:
        yield 3
    except Malfunction:
        print 'malfunction!'

    yield 4

    print 'done'

gen = my_generator()

print gen.next() # start the generator
print gen.send(10) # send the value 10
print gen.send(20) # send the value 20
print gen.throw(Malfunction()) # raise an exception inside the generator

try:
    gen.next()
except StopIteration:
    pass
```

In Python 2.5 and later versions, the `yield` statement is an expression that evaluates to a value. And the code that restarts the generator can determine that value using the `send` method instead of `next` (if you use `next` the value is `None`). What's more, you can actually raise an arbitrary exception *inside* the generator using the `throw` method. How cool is that?

Inline Callbacks

Given what we just reviewed about `send`ing and `throw`ing values and exceptions into a generator, we can envision a generator as a series of callbacks, like the ones in a deferred, which receive either results or failures. The callbacks are separated by `yield`s and the value of each `yield` expression is the result for the next callback (or the `yield` raises an exception and that's the failure). Figure 35 shows the correspondence:

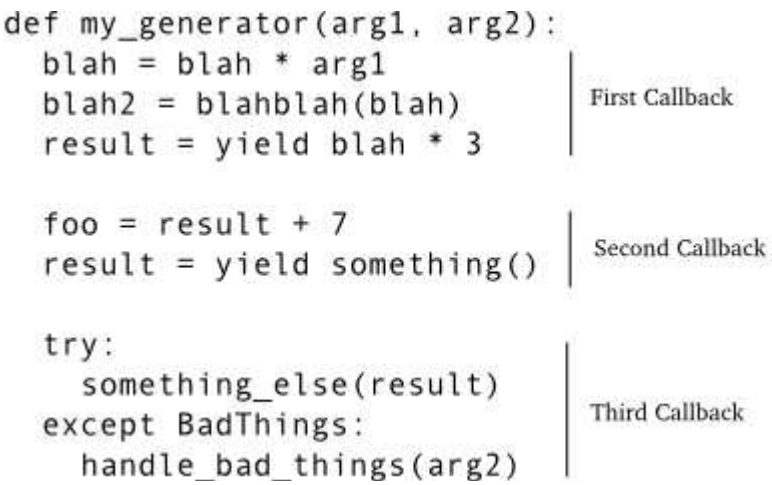

Figure 35: generator as a callback sequence

Now when a series of callbacks is chained together in a deferred, each callback receives the result from the one prior. That's easy enough to do with a generator — just `send` the value you got from the previous run of the generator (the value it `yield`ed) the next time you restart it. But that also seems a bit silly. Since the generator computed the value to begin with, why bother sending it back? The generator could just save the value in a variable for the next time it's needed. So what's the point?

Recall the fact we learned in Part 13, that the callbacks in a deferred can return deferreds themselves. And when that happens, the outer deferred is paused until the inner deferred fires, and then the next callback (or errback) in the outer deferred's chain is called with the result (or failure) from the inner deferred.

So imagine that our generator `yield`s a deferred object instead of an ordinary Python value. The generator is now "paused", and that's automatic; generators always pause after every `yield` statement until they are explicitly restarted. So we can delay restarting the generator until the deferred fires, at which point we either `send` the value (if the deferred succeeds) or `throw` the exception (if the deferred fails). That would make our generator a genuine sequence of asynchronous callbacks and that's the idea behind the `inlineCallbacks` function in `twisted.internet.defer`.

inlineCallbacks

Consider the example program in `inline-callbacks/inline-callbacks-1.py`:

```
 1 from twisted.internet.defer import inlineCallbacks, Deferred
 2
 3 @inlineCallbacks
 4 def my_callbacks():
 5     from twisted.internet import reactor
 6
 7     print 'first callback'
 8     result = yield 1 # yielded values that aren't deferred come right back
 9
10     print 'second callback got', result
11     d = Deferred()
12     reactor.callLater(5, d.callback, 2)
13     result = yield d # yielded deferreds will pause the generator
14
15     print 'third callback got', result # the result of the deferred
16
17     d = Deferred()
18     reactor.callLater(5, d.errback, Exception(3))
19
20     try:
21         yield d
```

```
22      except Exception, e:
23          result = e
24
25      print 'fourth callback got', repr(result) # the exception from the deferrec
26
27      reactor.stop()
28
29  from twisted.internet import reactor
30  reactor.callWhenRunning(my_callbacks)
31  reactor.run()
```

Run the example and you will see the generator execute to the end and then stop the reactor. The example illustrates several aspects of the `inlineCallbacks` function. First, `inlineCallbacks` is a decorator and it always decorates generator functions, i.e., functions that use `yield`. The whole purpose of `inlineCallbacks` is turn a generator into a series of asynchronous callbacks according to the scheme we outlined before.

Second, when we invoke an `inlineCallbacks`-decorated function, we don't need to call `next` or `send` or `throw` ourselves. The decorator takes care of those details for us and ensures the generator will run to the end (assuming it doesn't raise an exception).

Third, if we `yield` a non-deferred value from the generator, it is immediately restarted with that same value as the result of the `yield`.

And finally, if we `yield` a deferred from the generator, it will not be restarted until that deferred fires. If the deferred succeeds, the result of the `yield` is just the result from the deferred. And if the deferred fails, the `yield` statement raises the exception. Note the exception is just an ordinary `Exception` object, rather than a `Failure`, and we can catch it with a `try/except` statement around the `yield` expression.

In the example we are just using `callLater` to fire the deferreds after a short period of time. While that's a handy way to put in a non-blocking delay into our callback chain, normally we would be `yield`ing a deferred returned by some other asynchronous operation (i.e., `get_poetry`) invoked from our generator.

Ok, now we know how an `inlineCallbacks`-decorated function runs, but what return value do you get if you actually call one? As you might have guessed, you get a deferred. Since we can't know exactly when that generator will stop running (it might `yield` one or more deferreds), the decorated function itself is asynchronous and a deferred is the appropriate return value. Note the deferred that is returned isn't one of the deferreds the generator may `yield`. Rather, it's a deferred that fires only after the generator has completely finished (or throws an exception).

If the generator throws an exception, the returned deferred will fire its errback chain with that exception wrapped in a `Failure`. But if we want the generator to return a normal value, we must "return" it using the `defer.returnValue` function. Like the ordinary `return` statement, it will also stop the generator (it actually raises a special exception). The `inline-callbacks/inline-callbacks-2.py` example illustrates both possibilities.

Client 7.0

Let's put `inlineCallbacks` to work with a new version of our poetry client. You can see the code in `twisted-client-7/get-poetry.py`. You may wish to compare it to client 6.0 in `twisted-client-6/get-poetry.py`. The relevant changes are in `poetry_main`:

```
1   def poetry_main():
2       addresses = parse_args()
3
4       xform_addr = addresses.pop(0)
5
6       proxy = TransformProxy(*xform_addr)
```

```
 7
 8       from twisted.internet import reactor
 9
10       results = []
11
12       @defer.inlineCallbacks
13       def get_transformed_poem(host, port):
14           try:
15               poem = yield get_poetry(host, port)
16           except Exception, e:
17               print >>sys.stderr, 'The poem download failed:', e
18               raise
19
20           try:
21               poem = yield proxy.xform('cummingsify', poem)
22           except Exception:
23               print >>sys.stderr, 'Cummingsify failed!'
24
25           defer.returnValue(poem)
26
27       def got_poem(poem):
28           print poem
29
30       def poem_done(_):
31           results.append(_)
32           if len(results) == len(addresses):
33               reactor.stop()
34
35       for address in addresses:
36           host, port = address
37           d = get_transformed_poem(host, port)
38           d.addCallbacks(got_poem)
39           d.addBoth(poem_done)
40
41       reactor.run()
```

In our new version the `inlineCallbacks` generator function `get_transformed_poem` is responsible for both fetching the poem and then applying the transformation (via the transform service). Since both operations are asynchronous, we yield a deferred each time and then (implicitly) wait for the result. As in client 6.0, if the transformation fails we just return the original poem. Notice we can use `try`/`except` statements to handle asynchronous errors inside the generator.

We can test the new client out in the same way as before. First start up a transform server:

```
python twisted-server-1/tranformedpoetry.py --port 10001
```

Then start a couple of poetry servers:

```
python twisted-server-1/fastpoetry.py --port 10002 poetry/fascination.txt
python twisted-server-1/fastpoetry.py --port 10003 poetry/science.txt
```

Now you can run the new client:

```
python twisted-client-7/get-poetry.py 10001 10002 10003
```

Try turning off one or more of the servers to see how the client handles errors.

Discussion

Like the `Deferred` object, the `inlineCallbacks` function gives us a new way of organizing our asynchronous callbacks. And, as with deferreds, `inlineCallbacks` doesn't change the rules of the game. Specifically, our callbacks still run one at a time, and they are still invoked by the reactor. We can confirm

that fact in our usual way by printing out a traceback from an inline callback, as in the example script `inline-callbacks/inline-callbacks-tb.py`. Run that code and you will get a traceback with `reactor.run()` at the top, lots of helper functions in between, and our callback at the bottom.

We can adapt Figure 29, which explains what happens when one callback in a deferred returns another deferred, to show what happens when an `inlineCallbacks` generator `yield`s a deferred. See Figure 36:

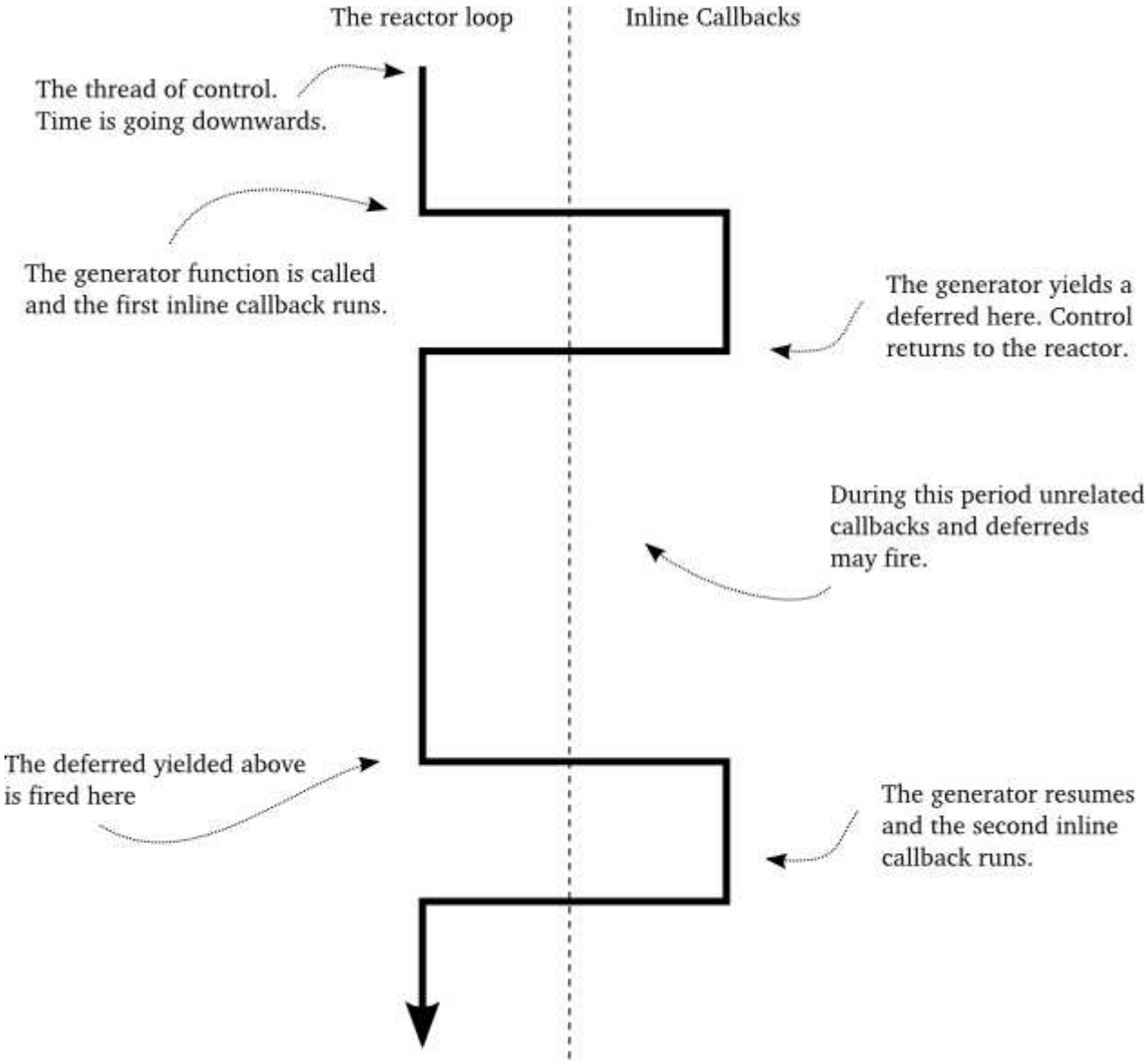

Figure 36: flow control in an inlineCallbacks function

The same figure works in both cases because the idea being illustrated is the same — one asynchronous operation is waiting for another.

Since `inlineCallbacks` and deferreds solve many of the same problems, why choose one over the other? Here are some potential advantages of `inlineCallbacks`:

- Since the callbacks share a namespace, there is no need to pass extra state around.
- The callback order is easier to see, as they just execute from top to bottom.
- With no function declarations for individual callbacks and implicit flow-control, there is generally less typing.
- Errors are handled with the familiar `try/except` statement.

And here are some potential pitfalls:

- The callbacks inside the generator cannot be invoked individually, which could make code re-use difficult. With a deferred, the code constructing the deferred is free to add arbitrary callbacks in an arbitrary order.
- The compact form of a generator can obscure the fact that an asynchronous callback is even involved. Despite its visually similar appearance to an ordinary sequential function, a generator behaves in a very different manner. The `inlineCallbacks` function is not a way to avoid learning the asynchronous programming model.

As with any technique, practice will provide the experience necessary to make an informed choice.

Summary

In this Part we learned about the `inlineCallbacks` decorator and how it allows us to express a sequence of asynchronous callbacks in the form of a Python generator.

In Part 18 we will learn a technique for managing a set of "parallel" asynchronous operations.

Suggested Exercises

1. Why is the `inlineCallbacks` function plural?
2. Study the implementation of `inlineCallbacks` and its helper function `_inlineCallbacks`. Ponder the phrase "the devil is in the details".
3. How many callbacks are contained in a generator with **N** `yield` statements, assuming it has no loops or `if` statements?
4. Poetry client 7.0 might have three generators running at once. Conceptually, how many different ways might they be interleaved with one another? Considering the way they are invoked in the poetry client and the implementation of `inlineCallbacks`, how many ways do you think are actually possible?
5. Move the `got_poem` callback in client 7.0 inside the generator.
6. Then move the `poem_done` callback inside the generator. Be careful! Make sure to handle all the failure cases so the reactor gets shutdown no matter what. How does the resulting code compare to using a deferred to shutdown the reactor?
7. A generator with `yield` statements inside a `while` loop can represent a conceptually infinite sequence. What does such a generator decorated with `inlineCallbacks` represent?

Part 18: Deferreds En Masse

This continues the introduction started here. You can find an index to the entire series here.

Introduction

In the last Part we learned a new way of structuring sequential asynchronous callbacks using a generator. Thus, including deferreds, we now have two techniques for chaining asynchronous operations together.

Sometimes, though, we want to run a group of asynchronous operations in "parallel". Since Twisted is single-threaded they won't really run concurrently, but the point is we want to use asynchronous I/O to work on a group of tasks as fast as possible. Our poetry clients, for example, download poems from multiple servers at the same time, rather than one server after another. That was the whole point of using Twisted for getting poetry, after all.

And, as a result, all our poetry clients have had to solve this problem: how do you know when all the asynchronous operations you have started are done? So far we have solved this by collecting our results into a list (like the `results` list in client 7.0) and checking the length of the list. We have to be careful to collect failures as well as successful results, otherwise a single failure will cause the program to run

forever, thinking there's still work left to do.

As you might expect, Twisted includes an abstraction you can use to solve this problem and we're going to take a look at it today.

The DeferredList

The `DeferredList` class allows us to treat a list of deferred objects as a single deferred. That way we can start a bunch of asynchronous operations and get notified only when all of them have finished (regardless of whether they succeeded or failed). Let's look at some examples.

In `deferred-list/deferred-list-1.py` you will find this code:

```
1  from twisted.internet import defer
2
3  def got_results(res):
4      print 'We got:', res
5
6  print 'Empty List.'
7  d = defer.DeferredList([])
8  print 'Adding Callback.'
9  d.addCallback(got_results)
```

And if you run it, you will get this output:

```
Empty List.
Adding Callback.
We got: []
```

Some things to notice:

- A `DeferredList` is created from a Python `list`. In this case the list is empty, but we'll soon see that the list elements must all be `Deferred` objects.
- A `DeferredList` is itself a deferred (it inherits from `Deferred`). That means you can add callbacks and errbacks to it just like you would a regular deferred.
- In the example above, our callback was fired as soon as we added it, so the `DeferredList` must have fired right away. We'll discuss that more in a second.
- The result of the deferred list was itself a list (empty).

Now look at `deferred-list/deferred-list-2.py`:

```
1  from twisted.internet import defer
2
3  def got_results(res):
4      print 'We got:', res
5
6  print 'One Deferred.'
7  d1 = defer.Deferred()
8  d = defer.DeferredList([d1])
9  print 'Adding Callback.'
10 d.addCallback(got_results)
11 print 'Firing d1.'
12 d1.callback('d1 result')
```

Now we are creating our `DeferredList` with a 1-element list containing a single deferred. Here's the output we get:

```
One Deferred.
Adding Callback.
Firing d1.
We got: [(True, 'd1 result')]
```

More things to notice:

- This time the `DeferredList` didn't fire its callback until we fired the deferred in the list.
- The result is still a list, but now it has one element.
- The element is a tuple whose second value is the result of the deferred in the list.

Let's try putting two deferreds in the list (deferred-list/deferred-list-3.py):

```
 1 from twisted.internet import defer
 2
 3 def got_results(res):
 4     print 'We got:', res
 5
 6 print 'Two Deferreds.'
 7 d1 = defer.Deferred()
 8 d2 = defer.Deferred()
 9 d = defer.DeferredList([d1, d2])
10 print 'Adding Callback.'
11 d.addCallback(got_results)
12 print 'Firing d1.'
13 d1.callback('d1 result')
14 print 'Firing d2.'
15 d2.callback('d2 result')
```

And here's the output:

```
Two Deferreds.
Adding Callback.
Firing d1.
Firing d2.
We got: [(True, 'd1 result'), (True, 'd2 result')]
```

At this point it's pretty clear the result of a `DeferredList`, at least for the way we've been using it, is a list with the same number of elements as the list of deferreds we passed to the constructor. And the elements of that result list contain the results of the original deferreds, at least if the deferreds succeed. That means the `DeferredList` itself doesn't fire until all the deferreds in the original list have fired. And a `DeferredList` created with an empty list fires right away since there aren't any deferreds to wait for.

What about the order of the results in the final list? Consider deferred-list/deferred-list-4.py:

```
 1 from twisted.internet import defer
 2
 3 def got_results(res):
 4     print 'We got:', res
 5
 6 print 'Two Deferreds.'
 7 d1 = defer.Deferred()
 8 d2 = defer.Deferred()
 9 d = defer.DeferredList([d1, d2])
10 print 'Adding Callback.'
11 d.addCallback(got_results)
12 print 'Firing d2.'
13 d2.callback('d2 result')
14 print 'Firing d1.'
15 d1.callback('d1 result')
```

Now we are firing `d2` first and then `d1`. Note the deferred list is still constructed with `d1` and `d2` in their original order. Here's the output:

```
Two Deferreds.
Adding Callback.
Firing d2.
Firing d1.
```

```
We got: [(True, 'd1 result'), (True, 'd2 result')]
```

The output list has the results in the same order as the original list of deferreds, not the order those deferreds happened to fire in. Which is very nice, because we can easily associate each individual result with the operation that generated it (for example, which poem came from which server).

Alright, what happens if one or more of the deferreds in the list fails? And what are those `True` values doing there? Let's try the example in deferred-list/deferred-list-5.py:

```
 1  from twisted.internet import defer
 2
 3  def got_results(res):
 4      print 'We got:', res
 5
 6  d1 = defer.Deferred()
 7  d2 = defer.Deferred()
 8  d = defer.DeferredList([d1, d2], consumeErrors=True)
 9  d.addCallback(got_results)
10  print 'Firing d1.'
11  d1.callback('d1 result')
12  print 'Firing d2 with errback.'
13  d2.errback(Exception('d2 failure'))
```

Now we are firing `d1` with a normal result and `d2` with an error. Ignore the `consumerErrors` option for now, we'll get back to it. Here's the output:

```
Firing d1.
Firing d2 with errback.
We got: [(True, 'd1 result'), (False, <twisted.python.failure.Failure <type 'exception
```

Now the tuple corresponding to `d2` has a `Failure` in slot two, and `False` in slot one. At this point it should be pretty clear how a `DeferredList` works (but see the Discussion below):

- A `DeferredList` is constructed with a list of deferred objects.
- A `DeferredList` is itself a deferred whose result is a list of the same length as the list of deferreds.
- The `DeferredList` fires after all the deferreds in the original list have fired.
- Each element of the result list corresponds to the deferred in the same position as the original list. If that deferred succeeded, the element is `(True, result)` and if the deferred failed, the element is `(False, failure)`.
- A `DeferredList` never fails, since the result of each individual deferred is collected into the list no matter what (but again, see the Discussion below).

Now let's talk about that `consumeErrors` option we passed to the `DeferredList`. If we run the same code but without passing the option (deferred-list/deferred-list-6.py), we get this output:

```
Firing d1.
Firing d2 with errback.
We got: [(True, 'd1 result'), (False, >twisted.python.failure.Failure >type 'exception
Unhandled error in Deferred:
Traceback (most recent call last):
Failure: exceptions.Exception: d2 failure
```

If you recall, the "Unhandled error in Deferred" message is generated when a deferred is garbage collected and the last callback in that deferred failed. The message is telling us we haven't caught all the potential asynchronous failures in our program. So where is it coming from in our example? It's clearly not coming from the `DeferredList`, since that succeeds. So it must be coming from `d2`.

A `DeferredList` needs to know when each deferred it is monitoring fires. And the `DeferredList` does that in the usual way — by adding a callback and errback to each deferred. And by default, the callback (and errback) return the original result (or failure) after putting it in the final list. And since returning the

original failure from the errback triggers the next errback, `d2` remains in the failed state after it fires.

But if we pass `consumeErrors=True` to the `DeferredList`, the errback added by the `DeferredList` to each deferred will instead return `None`, thus "consuming" the error and eliminating the warning message. We could also handle the error by adding our own errback to `d2`, as in `deferred-list/deferred-list-7.py`.

Client 8.0

Version 8.0 of our Get Poetry Now! client uses a `DeferredList` to find out when all the poetry has finished (or failed). You can find the new client in `twisted-client-8/get-poetry.py`. Once again the only change is in `poetry_main`. Let's look at the important changes:

```
1  ...
2  ds = []
3
4  for (host, port) in addresses:
5      d = get_transformed_poem(host, port)
6      d.addCallbacks(got_poem)
7      ds.append(d)
8
9  dlist = defer.DeferredList(ds, consumeErrors=True)
10 dlist.addCallback(lambda res : reactor.stop())
```

You may wish to compare it to the same section of `client 7.0`.

In client 8.0, we don't need the `poem_done` callback or the `results` list. Instead, we put each deferred we get back from `get_transformed_poem` into a list (`ds`) and then create a `DeferredList`. Since the `DeferredList` won't fire until all the poems have finished or failed, we just add a callback to the `DeferredList` to shutdown the reactor. In this case, we aren't using the result from the `DeferredList`, we just need to know when everything is finished. And that's it!

Discussion

We can visualize how a `DeferredList` works in Figure 37:

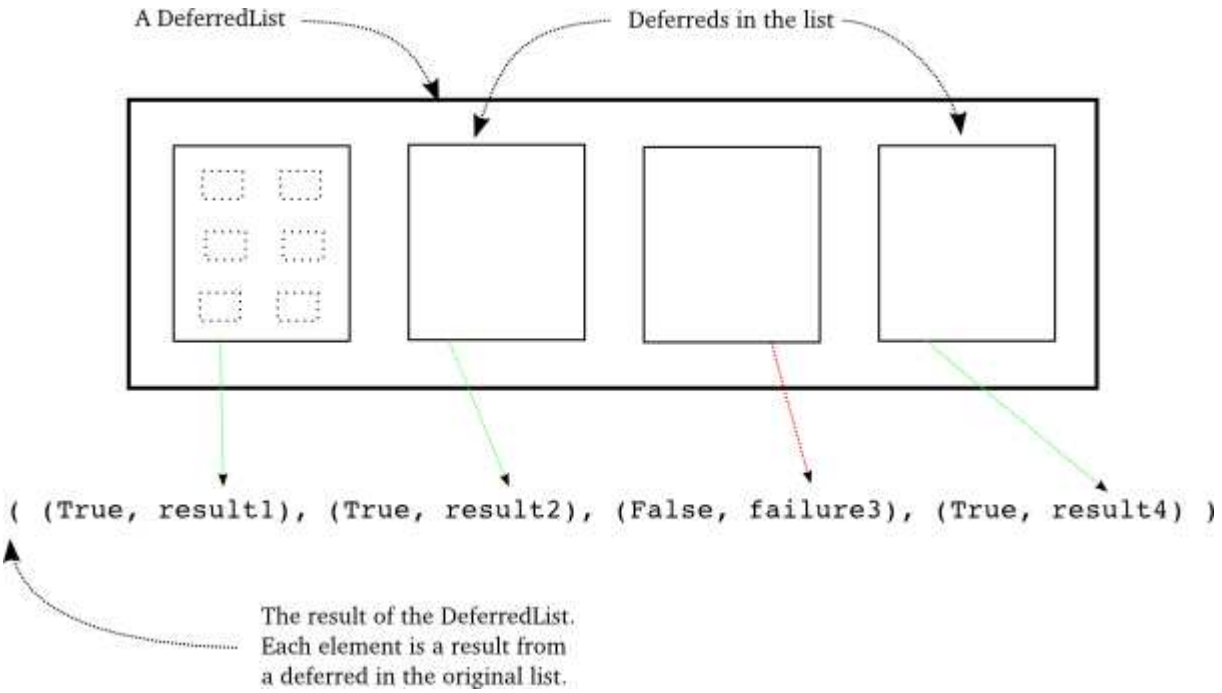

Figure 37: the result of a DeferredList

Pretty simple, really. There are a couple options to `DeferredList` we haven't covered, and which change the behavior from what we have described above. We will leave them for you to explore in the Exercises below.

In the next Part we will cover one more feature of the `Deferred` class, a feature recently introduced in Twisted 10.1.0.

Suggested Exercises

1. Read the source code for the `DeferredList`.
2. Modify the examples in `deferred-list` to experiment with the optional constructor arguments `fireOnOneCallback` and `fireOnOneErrback`. Come up with scenarios where you would use one or the other (or both).
3. Can you create a `DeferredList` using a list of `DeferredList`s? If so, what would the result look like?
4. Modify client 8.0 so that it doesn't print out anything until all the poems have finished downloading. This time you will use the result from the `DeferredList`.
5. Define the semantics of a `DeferredDict` and then implement it.

Part 19: I Thought I Wanted It But I Changed My Mind

This continues the introduction started here. You can find an index to the entire series here.

Introduction

Twisted is an ongoing project and the Twisted developers regularly add new features and extend old ones. With the release of Twisted 10.1.0, the developers added a new capability — cancellation — to the `Deferred` class which we're going to investigate today.

Asynchronous programming decouples requests from responses and thus raises a new possibility: between asking for the result and getting it back you might decide you don't want it anymore. Consider the poetry proxy server from Part 14. Here's how the proxy worked, at least for the first request of a poem:

1. A request for a poem comes in.
2. The proxy contacts the real server to get the poem.
3. Once the poem is complete, send it to the original client.

Which is all well and good, but what if the client hangs up before getting the poem? Maybe they requested the complete text of Paradise Lost and then decided they really wanted a haiku by Kojo. Now our proxy is stuck with downloading the first one and that slow server is going to take a while. Better to close the connection and let the slow server go back to sleep.

Recall Figure 15, a diagram that shows the conceptual flow of control in a synchronous program. In that figure we see function calls going down, and exceptions going back up. If we wanted to cancel a synchronous function call (and this is just hypothetical) the flow control would go in the same direction as the function call, from high-level code to low-level code as in Figure 38:

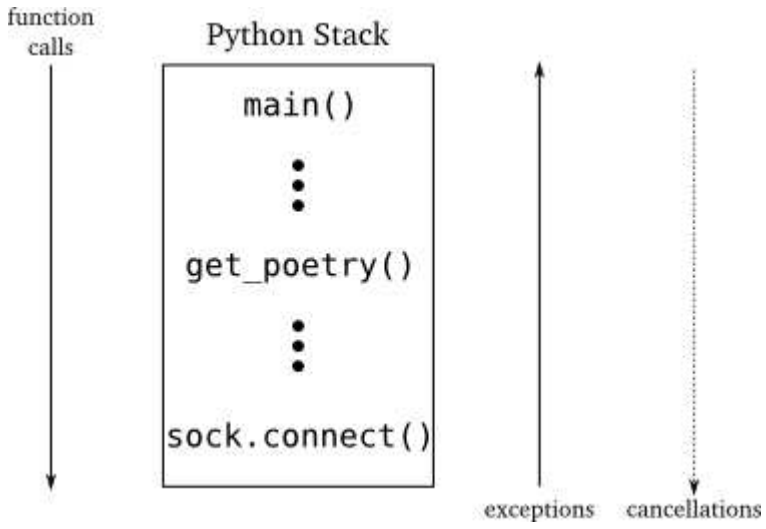

Figure 38: synchronous program flow, with hypothetical cancellation

Of course, in a synchronous program that isn't possible because the high-level code doesn't even resume running until the low-level operation is finished, at which point there is nothing to cancel. But in an asynchronous program the high-level code gets control of the program before the low-level code is done, which at least raises the possibility of canceling the low-level request before it finishes.

In a Twisted program, the lower-level request is embodied by a `Deferred` object, which you can think of as a "handle" on the outstanding asynchronous operation. The normal flow of information in a deferred is downward, from low-level code to high-level code, which matches the flow of return information in a synchronous program. Starting in Twisted 10.1.0, high-level code can send information back the other direction — it can tell the low-level code it doesn't want the result anymore. See Figure 39:

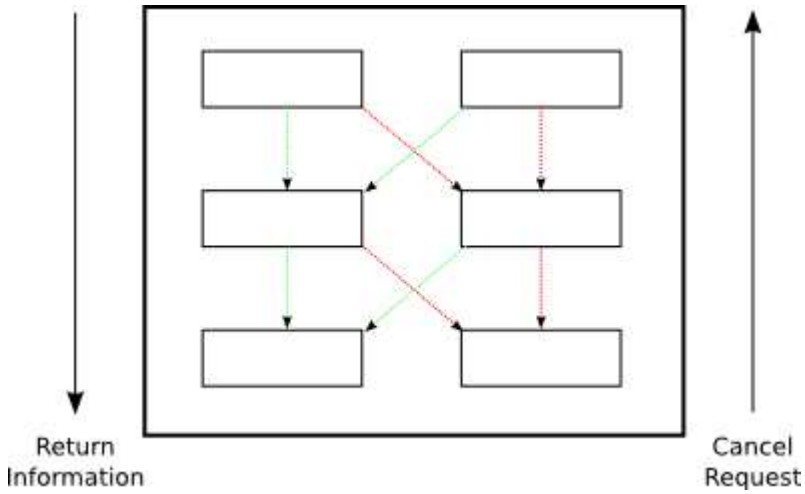

Figure 39: Information flow in a deferred, including cancellation

Canceling Deferreds

Let's take a look at a few sample programs to see how canceling deferreds actually works. Note, to run the examples and other code in this Part you will need a version of Twisted 10.1.0 or later. Consider `deferred-cancel/defer-cancel-1.py`:

```
1 from twisted.internet import defer
2
3 def callback(res):
4     print 'callback got:', res
```

```
5
6  d = defer.Deferred()
7  d.addCallback(callback)
8  d.cancel()
9  print 'done'
```

With the new cancellation feature, the `Deferred` class got a new method called `cancel`. The example code makes a new deferred, adds a callback, and then cancels the deferred without firing it. Here's the output:

```
done
Unhandled error in Deferred:
Traceback (most recent call last):
Failure: twisted.internet.defer.CancelledError:
```

Ok, so canceling a deferred appears to cause the errback chain to run, and our regular callback is never called at all. Also notice the error is a `twisted.internet.defer.CancelledError`, a custom Exception that means the deferred was canceled (but keep reading!). Let's try adding an errback in deferred-cancel/defer-cancel-2.py:

```
1  from twisted.internet import defer
2
3  def callback(res):
4      print 'callback got:', res
5
6  def errback(err):
7      print 'errback got:', err
8
9  d = defer.Deferred()
10 d.addCallbacks(callback, errback)
11 d.cancel()
12 print 'done'
```

Now we get this output:

```
errback got: [Failure instance: Traceback (failure with no frames): <class 'twisted.in
]
done
```

So we can 'catch' the errback from a cancel just like any other deferred failure.

Ok, let's try firing the deferred and then canceling it, as in deferred-cancel/defer-cancel-3.py:

```
1  from twisted.internet import defer
2
3  def callback(res):
4      print 'callback got:', res
5
6  def errback(err):
7      print 'errback got:', err
8
9  d = defer.Deferred()
10 d.addCallbacks(callback, errback)
11 d.callback('result')
12 d.cancel()
13 print 'done'
```

Here we fire the deferred normally with the `callback` method and then cancel it. Here's the output:

```
callback got: result
done
```

Our callback was invoked (just as we would expect) and then the program finished normally, as if `cancel`

was never called at all. So it seems canceling a deferred has no effect if it has already fired (but keep reading!).

What if we fire the deferred *after* we cancel it, as in deferred-cancel/defer-cancel-4.py?

```
1  from twisted.internet import defer
2
3  def callback(res):
4      print 'callback got:', res
5
6  def errback(err):
7      print 'errback got:', err
8
9  d = defer.Deferred()
10 d.addCallbacks(callback, errback)
11 d.cancel()
12 d.callback('result')
13 print 'done'
```

In that case we get this output:

```
errback got: [Failure instance: Traceback (failure with no frames): <class 'twisted.in
]
done
```

Interesting! That's the same output as the second example, where we never fired the deferred at all. So if the deferred has been canceled, firing the deferred normally has no effect. But why doesn't `d.callback('result')` raise an error, since you're not supposed to be able to fire a deferred more than once, and the errback chain has clearly run?

Consider Figure 39 again. Firing a deferred with a result or failure is the job of lower-level code, while canceling a deferred is an action taken by higher-level code. Firing the deferred means "Here's your result", while canceling a deferred means "I don't want it any more". And remember that canceling is a new feature, so most existing Twisted code is not written to handle cancel operations. But the Twisted developers have made it possible for us to cancel any deferred we want to, even if the code we got the deferred from was written before Twisted 10.1.0.

To make that possible, the `cancel` method actually does two things:

1. Tell the `Deferred` object *itself* that you don't want the result if it hasn't shown up yet (i.e, the deferred hasn't been fired), and thus to ignore any subsequent invocation of `callback` or `errback`.
2. And, *optionally*, tell the lower-level code that is producing the result to take whatever steps are required to cancel the operation.

Since older Twisted code is going to go ahead and fire that canceled deferred anyway, step #1 ensures our program won't blow up if we cancel a deferred we got from an older library.

This means we are always free to cancel a deferred, and we'll be sure not to get the result if it hasn't arrived (even if it arrives later). But canceling the deferred might not actually cancel the asynchronous operation. Aborting an asynchronous operation requires a context-specific action. You might need to close a network connection, roll back a database transaction, kill a sub-process, et cetera. And since a deferred is just a general-purpose callback organizer, how is it supposed to know what specific action to take when you cancel it? Or, alternatively, how could it forward the cancel request to the lower-level code that created and returned the deferred in the first place? Say it with me now:

> I know, with a callback!

Canceling Deferreds, Really

Alright, take a look at deferred-cancel/defer-cancel-5.py:

```
1  from twisted.internet import defer
2
3  def canceller(d):
4      print "I need to cancel this deferred:", d
5
6  def callback(res):
7      print 'callback got:', res
8
9  def errback(err):
10     print 'errback got:', err
11
12 d = defer.Deferred(canceller) # created by lower-level code
13 d.addCallbacks(callback, errback) # added by higher-level code
14 d.cancel()
15 print 'done'
```

This code is basically like the second example, except there is a third callback (`canceller`) that's passed to the `Deferred` when we create it, rather than added afterwards. This callback is in charge of performing the context-specific actions required to abort the asynchronous operation (only if the deferred is actually canceled, of course). The `canceller` callback is necessarily part of the lower-level code that returns the deferred, not the higher-level code that receives the deferred and adds its own callbacks and errbacks.

Running the example produces this output:

```
I need to cancel this deferred: <Deferred at 0xb7669d2cL>
errback got: [Failure instance: Traceback (failure with no frames): <class 'twisted.in
]
done
```

As you can see, the `canceller` callback is given the deferred whose result we no longer want. That's where we would take whatever action we need to in order to abort the asynchronous operation. Notice that `canceller` is invoked before the errback chain fires. In fact, we may choose to fire the deferred ourselves at this point with any result or error of our choice (and thus preempting the `CancelledError` failure). Both possibilities are illustrated in deferred-cancel/defer-cancel-6.py and deferred-cancel/defer-cancel-7.py.

Let's do one more simple test before we fire up the reactor. We'll create a deferred with a `canceller` callback, fire it normally, and then cancel it. You can see the code in deferred-cancel/defer-cancel-8.py. By examining the output of that script, you can see that canceling a deferred after it has been fired does *not* invoke the `canceller` callback. And that's as we would expect since there's nothing to cancel.

The examples we've looked at so far haven't had any actual asynchronous operations. Let's make a simple program that invokes one asynchronous operation, then we'll figure out how to make that operation cancellable. Consider the code in deferred-cancel/defer-cancel-9.py:

```
1  from twisted.internet.defer import Deferred
2
3  def send_poem(d):
4      print 'Sending poem'
5      d.callback('Once upon a midnight dreary')
6
7  def get_poem():
8      """Return a poem 5 seconds later."""
9      from twisted.internet import reactor
10     d = Deferred()
11     reactor.callLater(5, send_poem, d)
12     return d
13
14 def got_poem(poem):
```

```
15      print 'I got a poem:', poem
16
17  def poem_error(err):
18      print 'get_poem failed:', err
19
20  def main():
21      from twisted.internet import reactor
22      reactor.callLater(10, reactor.stop) # stop the reactor in 10 seconds
23
24      d = get_poem()
25      d.addCallbacks(got_poem, poem_error)
26
27      reactor.run()
28
29  main()
```

This example includes a `get_poem` function that uses the reactor's `callLater` method to asynchronously return a poem five seconds after `get_poem` is called. The `main` function calls `get_poem`, adds a callback/errback pair, and then starts up the reactor. We also arrange (again using `callLater`) to stop the reactor in ten seconds. Normally we would do this by attaching a callback to the deferred, but you'll see why we do it this way shortly.

Running the example produces this output (after the appropriate delay):

```
Sending poem
I got a poem: Once upon a midnight dreary
```

And after ten seconds our little program comes to a stop. Now let's try canceling that deferred before the poem is sent. We'll just add this bit of code to cancel the deferred after two seconds (well before the five second delay on the poem itself):

```
    reactor.callLater(2, d.cancel) # cancel after 2 seconds
```

The complete program is in [deferred-cancel/defer-cancel-10.py](), which produces the following output:

```
get_poem failed: [Failure instance: Traceback (failure with no frames): <class 'twiste
]
Sending poem
```

This example clearly illustrates that canceling a deferred does not necessarily cancel the underlying asynchronous request. After two seconds we see the output from our errback, printing out the `CancelledError` as we would expect. But then after five seconds will still see the output from `send_poem` (but the callback on the deferred doesn't fire).

At this point we're just in the same situation as [deferred-cancel/defer-cancel-4.py](). "Canceling" the deferred causes the eventual result to be ignored, but doesn't abort the operation in any real sense. As we learned above, to make a truly cancelable deferred we must add a `cancel` callback when the deferred is created.

What does this new callback need to do? Take a look at the documentation for the `callLater` method. The return value of `callLater` is another object, implementing `IDelayedCall`, with a `cancel` method we can use to prevent the delayed call from being executed.

That's pretty simple, and the updated code is in [deferred-cancel/defer-cancel-11.py](). The relevant changes are all in the `get_poem` function:

```
1   def get_poem():
2       """Return a poem 5 seconds later."""
3
4       def canceler(d):
```

```
        # They don't want the poem anymore, so cancel the delayed call
        delayed_call.cancel()

        # At this point we have three choices:
        #   1. Do nothing, and the deferred will fire the errback
        #      chain with CancelledError.
        #   2. Fire the errback chain with a different error.
        #   3. Fire the callback chain with an alternative result.

    d = Deferred(canceler)

    from twisted.internet import reactor
    delayed_call = reactor.callLater(5, send_poem, d)

    return d
```

In this new version, we save the return value from `callLater` so we can use it in our cancel callback. The only thing our callback needs to do is invoke `delayed_call.cancel()`. But as we discussed above, we could also choose to fire the deferred ourselves. The latest version of our example produces this output:

```
get_poem failed: [Failure instance: Traceback (failure with no frames): <class 'twiste
]
```

As you can see, the deferred is canceled and the asynchronous operation has truly been aborted (i.e., we don't see the `print` output from `send_poem`).

Poetry Proxy 3.0

As we discussed in the Introduction, the poetry proxy server is a good candidate for implementing cancellation, as it allows us to abort the poem download if it turns out that nobody wants it (i.e., the client closes the connection before we send the poem). Version 3.0 of the proxy, located in `twisted-server-4/poetry-proxy.py`, implements deferred cancellation. The first change is in the `PoetryProxyProtocol`:

```
class PoetryProxyProtocol(Protocol):

    def connectionMade(self):
        self.deferred = self.factory.service.get_poem()
        self.deferred.addCallback(self.transport.write)
        self.deferred.addBoth(lambda r: self.transport.loseConnection())

    def connectionLost(self, reason):
        if self.deferred is not None:
            deferred, self.deferred = self.deferred, None
            deferred.cancel() # cancel the deferred if it hasn't fired
```

You might compare it to the older version. The two main changes are:

1. Save the deferred we get from `get_poem` so we can cancel later if we need to.
2. Cancel the deferred when the connection is closed. Note this also cancels the deferred after we actually get the poem, but as we discovered in the examples, canceling a deferred that has already fired has no effect.

Now we need to make sure that canceling the deferred actually aborts the poem download. For that we need to change the `ProxyService`:

```
class ProxyService(object):

    poem = None # the cached poem

    def __init__(self, host, port):
```

```
6           self.host = host
7           self.port = port
8
9       def get_poem(self):
10          if self.poem is not None:
11              print 'Using cached poem.'
12              # return an already-fired deferred
13              return succeed(self.poem)
14
15          def canceler(d):
16              print 'Canceling poem download.'
17              factory.deferred = None
18              connector.disconnect()
19
20          print 'Fetching poem from server.'
21          deferred = Deferred(canceler)
22          deferred.addCallback(self.set_poem)
23          factory = PoetryClientFactory(deferred)
24          from twisted.internet import reactor
25          connector = reactor.connectTCP(self.host, self.port, factory)
26          return factory.deferred
27
28      def set_poem(self, poem):
29          self.poem = poem
30          return poem
```

Again, you may wish to compare this with the older version. This class has a few more changes:

1. We save the return value from `reactor.connectTCP`, an IConnector object. We can use the `disconnect` method on that object to close the connection.
2. We create the deferred with a `canceler` callback. That callback is a closure which uses the `connector` to close the connection. But first it sets the `factory.deferred` attribute to `None`. Otherwise, the factory might fire the deferred with a "connection closed" errback before the deferred itself fires with a `CancelledError`. Since this deferred was canceled, having the deferred fire with `CancelledError` seems more explicit.

You might also notice we now create the deferred in the `ProxyService` instead of the `PoetryClientFactory`. Since the canceler callback needs to access the `IConnector` object, the `ProxyService` ends up being the most convenient place to create the deferred.

And, as in one of our earlier examples, our `canceler` callback is implemented as a closure. Closures seem to be very useful when implementing cancel callbacks!

Let's try out our new proxy. First start up a *slow* server. It needs to be slow so we actually have time to cancel:

```
python blocking-server/slowpoetry.py --port 10001 poetry/fascination.txt
```

Now we can start up our proxy (remember you need Twisted 10.1.0):

```
python twisted-server-4/poetry-proxy.py --port 10000 10001
```

Now we can start downloading a poem from the proxy using any client, or even just `curl`:

```
curl localhost:10000
```

After a few seconds, press `Ctrl-C` to stop the client, or the `curl` process. In the terminal running the proxy you should
see this output:

```
Fetching poem from server.
Canceling poem download.
```

And you should see the slow server has stopped printing output for each bit of poem it sends, since our proxy hung up. You can start and stop the client multiple times to verify each download is canceled each time. But if you let the poem run to completion, then the proxy caches the poem and sends it immediately after that.

One More Wrinkle

We said several times above that canceling an already-fired deferred has no effect. Well, that's not quite true. In Part 13 we learned that the callbacks and errbacks attached to a deferred may return deferreds themselves. And in that case, the original (outer) deferred pauses the execution of its callback chains and waits for the inner deferred to fire (see Figure 28).

Thus, even though a deferred has fired the higher-level code that made the asynchronous request may not have received the result yet, because the callback chain is paused waiting for an inner deferred to finish. So what happens if the higher-level code cancels that outer deferred? In that case the outer deferred does not cancel itself (it has already fired after all); instead, the outer deferred cancels the inner deferred.

So when you cancel a deferred, you might not be canceling the main asynchronous operation, but rather some other asynchronous operation triggered as a result of the first. Whew!

We can illustrate this with one more example. Consider the code in deferred-cancel/defer-cancel-12.py:

```
1  from twisted.internet import defer
2
3  def cancel_outer(d):
4      print "outer cancel callback."
5
6  def cancel_inner(d):
7      print "inner cancel callback."
8
9  def first_outer_callback(res):
10     print 'first outer callback, returning inner deferred'
11     return inner_d
12
13 def second_outer_callback(res):
14     print 'second outer callback got:', res
15
16 def outer_errback(err):
17     print 'outer errback got:', err
18
19 outer_d = defer.Deferred(cancel_outer)
20 inner_d = defer.Deferred(cancel_inner)
21
22 outer_d.addCallback(first_outer_callback)
23 outer_d.addCallbacks(second_outer_callback, outer_errback)
24
25 outer_d.callback('result')
26
27 # at this point the outer deferred has fired, but is paused
28 # on the inner deferred.
29
30 print 'canceling outer deferred.'
31 outer_d.cancel()
32
33 print 'done'
```

In this example we create two deferreds, the outer and the inner, and have one of the outer callbacks return the inner deferred. First we fire the outer deferred, and then we cancel it. The example produces this output:

```
first outer callback, returning inner deferred
canceling outer deferred.
inner cancel callback.
outer errback got: [Failure instance: Traceback (failure with no frames): <class 'twis
]
done
```

As you can see, canceling the outer deferred does not cause the outer cancel callback to fire. Instead, it cancels the inner deferred so the inner cancel callback fires, and then outer errback receives the `CancelledError` (from the inner deferred).

You may wish to stare at that code a while, and try out variations to see how they affect the outcome.

Discussion

Canceling a deferred can be a very useful operation, allowing our programs to avoid work they no longer need to do. And as we have seen, it can be a little bit tricky, too.

One very important fact to keep in mind is that canceling a deferred doesn't necessarily cancel the underlying asynchronous operation. In fact, as of this writing, most deferreds won't really "cancel", since most Twisted code was written prior to Twisted 10.1.0 and hasn't been updated. This includes many of the APIs in Twisted itself! Check the documentation and/or the source code to find out whether canceling the deferred will truly cancel the request, or simply ignore it.

And the second important fact is that simply returning a deferred from your asynchronous APIs will not necessarily make them cancelable in the complete sense of the word. If you want to implement canceling in your own programs, you should study the Twisted source code to find more examples. Cancellation is a brand new feature so the patterns and best practices are still being worked out.

Looking Ahead

At this point we've learned just about everything about Deferreds and the core concepts behind Twisted. Which means there's not much more to introduce, as the rest of Twisted consists mainly of specific applications, like web programming or asynchronous database access. So in the next couple of Parts we're going to take a little detour and look at two other systems that use asynchronous I/O to see how some of their ideas relate to the ideas in Twisted. Then, in the final Part, we will wrap up and suggest ways to continue your Twisted education.

Suggested Exercises

1. Did you know you can spell canceled with one or two els? It's true. It all depends on what sort of mood you're in.
2. Peruse the source code of the `Deferred` class, paying special attention to the implementation of cancellation.
3. Search the Twisted 10.10 source code for examples of deferreds with cancel callbacks. Study their implementation.
4. Make the deferred returned by the `get_poetry` method of one of our poetry clients cancelable.
5. Make a reactor-based example that illustrates canceling an outer deferred which is paused on an inner deferred. If you use `callLater` you will need to choose the delays carefully to ensure the outer deferred is canceled at the right moment.
6. Find an asynchronous API in Twisted that doesn't support a true cancel and implement cancellation for it. Submit a patch to the Twisted project. Don't forget unit tests!

Part 20: Wheels within Wheels: Twisted and Erlang

This continues the introduction started here. You can find an index to the entire series here.

Introduction

One fact we've uncovered in this series is that mixing synchronous "plain Python" code with asynchronous Twisted code is not a straightforward task, since blocking for an indeterminate amount of time in a Twisted program will eliminate many of the benefits you are trying to achieve using the asynchronous model.

If this is your first introduction to asynchronous programming it may seem as if the knowledge you have gained is of somewhat limited applicability. You can use these new techniques inside of Twisted, but not in the much larger world of general Python code. And when working with Twisted, you are generally limited to libraries written specifically for use as part of a Twisted program, at least if you want to call them directly from the thread running the reactor.

But asynchronous programming techniques have been around for quite some time and are hardly confined to Twisted. There are in fact a startling number of asynchronous programming frameworks in Python alone. A bit of searching around will probably yield a couple dozen of them. They differ from Twisted in their details, but the basic ideas (asynchronous I/O, processing data in small chunks across multiple data streams) are the same. So if you need, or choose, to use an alternative framework you will already have a head start having learned Twisted.

And moving outside of Python, there are plenty of other languages and systems that are either based around, or make use of, the asynchronous programming model. Your knowledge of Twisted will continue to serve you as you explore the wider areas of this subject.

In this Part we're going to take a very brief look at Erlang, a programming language and runtime system that makes extensive use of asynchronous programming concepts, but does so in a unique way. Please note this is not meant as a general introduction to Erlang. Rather, it is a short exploration of some of the ideas embedded in Erlang and how they connect with the ideas in Twisted. The basic theme is the knowledge you have gained learning Twisted can be applied when learning other technologies.

Callbacks Reimagined

Consider Figure 6, a graphical representation of a callback. The principle callback in Poetry Client 3.0, introduced in Part 6, and all subsequent poetry clients is the `dataReceived` method. That callback is invoked each time we get a bit more poetry from one of the poetry servers we have connected to.

Let's say our client is downloading three poems from three different servers. Looking at things from the point of view of the reactor (and that's the viewpoint we've emphasized the most in this series), we've got a single big loop which makes one or more callbacks each time it goes around. See Figure 40:

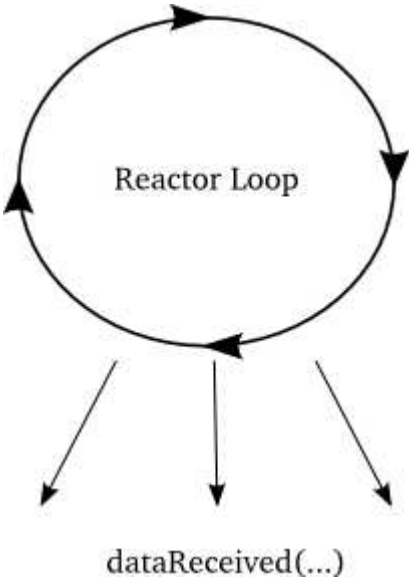

Figure 40: callbacks from the reactor viewpoint

This figure shows the reactor happily spinning around, calling `dataReceived` as the poetry comes in. Each invocation of `dataReceived` is applied to one particular instance of our `PoetryProtocol` class. And we know there are three instances because we are downloading three poems (and so there must be three connections).

Let's think about this picture from the point of view of *one* of those Protocol instances. Remember each Protocol is only concerned with a single connection (and thus a single poem). That instance "sees" a stream of method calls, each one bearing the next piece of the poem, like this:

```
1 dataReceived(self, "When I have fears")
2 dataReceived(self, " that I may cease to be")
3 dataReceived(self, "Before my pen has glea")
4 dataReceived(self, "n'd my teeming brain")
5 ...
```

While this isn't strictly speaking an actual Python loop, we can conceptualize it as one:

```
1 for data in poetry_stream(): # pseudo-code
2     dataReceived(data)
```

We can envision this "callback loop" in Figure 41:

Figure 41: A virtual callback loop

Again, this is not a `for` loop or a `while` loop. The only significant Python loop in our poetry clients is the reactor. But we can think of each Protocol as a virtual loop that ticks around once each time some poetry

for that particular poem comes in. With that in mind we can re-imagine the entire client in Figure 42:

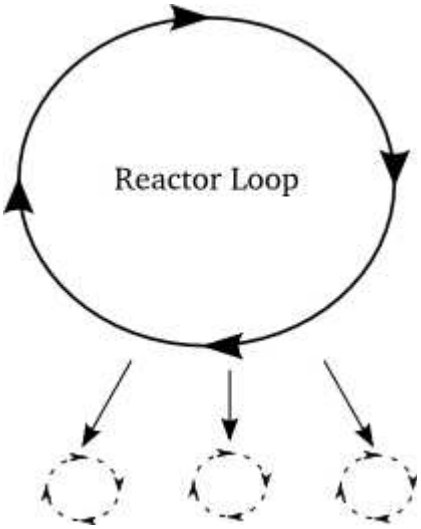

Figure 42: the reactor spinning some virtual loops

In this figure we have one big loop, the reactor, and three virtual loops, the individual poetry protocol instances. The big loop spins around and, in so doing, causes the virtual loops to tick over as well, like a set of interlocking gears.

Enter Erlang

Erlang, like Python, is a general purpose dynamically typed programming language originally created in the 80's. Unlike Python, Erlang is functional rather than object-oriented, and has a syntax reminiscent of Prolog, the language in which Erlang was originally implemented. Erlang was designed for building highly reliable distributed telephony systems, and thus Erlang contains extensive networking support.

One of Erlang's most distinctive features is a concurrency model involving lightweight processes. An Erlang process is neither an operating system process nor an operating system thread. Rather, it is an independently running function inside the Erlang runtime with its own stack. Erlang processes are not lightweight threads because Erlang processes cannot share state (and most data types are immutable anyway, Erlang being a functional programming language). An Erlang process can interact with other Erlang processes only by sending messages, and messages are always, at least conceptually, copied and never shared.

So an Erlang program might look like Figure 43:

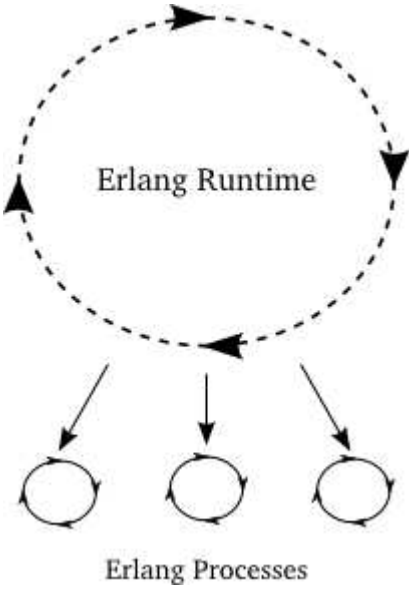

Figure 43: An Erlang program
with three processes

In this figure the individual processes have become "real", since processes are first-class constructs in Erlang, just like objects are in Python. And the runtime has become "virtual", not because it isn't there, but because it's not necessarily a simple loop. The Erlang runtime may be multi-threaded and, as it has to implement a full-blown programming language, it's in charge of a lot more than handling asynchronous I/O. Furthermore, a language runtime is not so much an extra construct, like the reactor in Twisted, as the medium in which the Erlang processes and code execute.

So an even better picture of an Erlang program might be Figure 44:

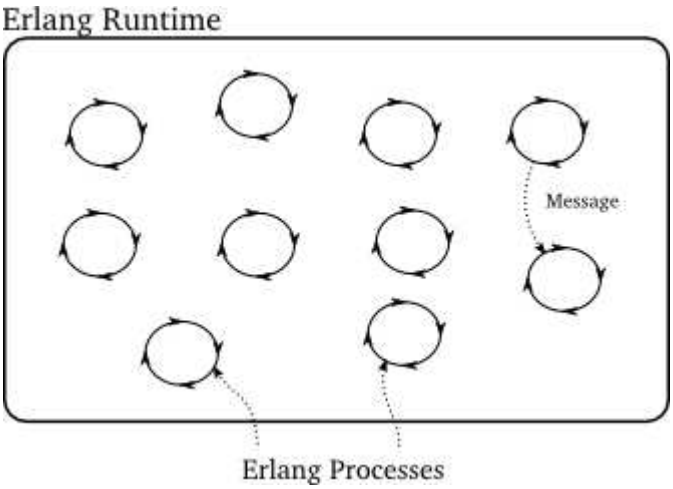

Figure 44: An Erlang program with several processes

Of course, the Erlang runtime does have to use asynchronous I/O and one or more select loops, because Erlang allows you to create *lots* of processes. Large Erlang programs can start tens or hundreds of thousands of Erlang processes, so allocating an actual OS thread to each one is simply out of the question. If Erlang is going to allow multiple processes to perform I/O, and still allow other processes to run even if that I/O blocks, then asynchronous I/O will have to be involved.

Note that our picture of an Erlang program has each process running "under its own power", rather than being spun around by callbacks. And that is very much the case. With the job of the reactor subsumed into the fabric of the Erlang runtime, the callback no longer has a central role to play. What would, in Twisted, be solved by using a callback would, in Erlang, be solved by sending an asynchronous message from one Erlang process to another.

An Erlang Poetry Client

Let's look at an Erlang poetry client. We're going to jump straight to a working version instead of building up slowly like we did with Twisted. Again, this isn't meant as a complete Erlang introduction. But if it piques your interest, we suggest some more in-depth reading at the end of this Part.

The Erlang client is listed in `erlang-client-1/get-poetry`. In order to run it you will, of course, need Erlang installed. Here's the code for the `main` function, which serves a similar purpose as the main functions in our Python clients:

```
1  main([]) ->
2      usage();
3
4  main(Args) ->
5      Addresses = parse_args(Args),
6      Main = self(),
7      [erlang:spawn_monitor(fun () -> get_poetry(TaskNum, Addr, Main) end)
8       || {TaskNum, Addr} <- enumerate(Addresses)],
9      collect_poems(length(Addresses), []).
```

If you've never seen Prolog or a similar language before then Erlang syntax is going to seem a little odd. But some people say that about Python, too. The main function is defined by two separate clauses, separated by a semicolon. Erlang chooses which clause to run by matching the arguments, so the first clause only runs if we execute the client without providing any command line arguments, and it just prints out a help message. The second clause is where all the action is.

Individual statements in an Erlang function are separated by commas, and all functions end with a period. Let's take each line in the second clause one at a time. The first line is just parsing the command line arguments and binding them to a variable (all variables in Erlang must be capitalized). The second line is using the Erlang `self` function to get the process ID of the currently running Erlang process (not OS process). Since this is the main function you can kind of think of it as the equivalent of the `__main__` module in Python. The third line is the most interesting:

```
1  [erlang:spawn_monitor(fun () -> get_poetry(TaskNum, Addr, Main) end)
2       || {TaskNum, Addr} <- enumerate(Addresses)],
```

This statement is an Erlang list comprehension, with a syntax similar to that in Python. It is spawning new Erlang processes, one for each poetry server we need to contact. And each process will run the same function (`get_poetry`) but with different arguments specific to that server. We also pass the PID of the main process so the new processes can send the poetry back (you generally need the PID of a process to send a message to it).

The last statement in `main` calls the `collect_poems` function which waits for the poetry to come back and for the `get_poetry` processes to finish. We'll look at the other functions in a bit, but first you might compare this Erlang
`main` function to the equivalent main in one of our Twisted clients.

Now let's look at the Erlang `get_poetry` function. There are actually two functions in our script called `get_poetry`. In Erlang, a function is identified by both name and arity, so our script contains two separate functions, `get_poetry/3` and `get_poetry/4` which accept three and four arguments respectively. Here's `get_poetry/3`, which is spawned by `main`:

```
1  get_poetry(Tasknum, Addr, Main) ->
2      {Host, Port} = Addr,
3      {ok, Socket} = gen_tcp:connect(Host, Port,
4                                     [binary, {active, false}, {packet, 0}]),
5      get_poetry(Tasknum, Socket, Main, []).
```

This function first makes a TCP connection, just like the Twisted client `get_poetry`. But then, instead of

returning, it proceeds to use that TCP connection by calling `get_poetry/4`, listed below:

```
1  get_poetry(Tasknum, Socket, Main, Packets) ->
2      case gen_tcp:recv(Socket, 0) of
3          {ok, Packet} ->
4              io:format("Task ~w: got ~w bytes of poetry from ~s\n",
5                        [Tasknum, size(Packet), peername(Socket)]),
6              get_poetry(Tasknum, Socket, Main, [Packet|Packets]);
7          {error, _} ->
8              Main ! {poem, list_to_binary(lists:reverse(Packets))}
9      end.
```

This Erlang function is doing the work of the `PoetryProtocol` from our Twisted client, except it does so using blocking function calls. The `gen_tcp:recv` function waits until some data arrives on the socket (or the socket is closed), however long that might be. But a "blocking" function in Erlang only blocks the process running the function, not the entire Erlang runtime. That TCP socket isn't really a blocking socket (you can't make a true blocking socket in pure Erlang code). For each of those Erlang sockets there is, somewhere inside the Erlang runtime, a "real" TCP socket set to non-blocking mode and used as part of a select loop.

But the Erlang process doesn't have to know about any of that. It just waits for some data to arrive and, if it blocks, some other Erlang process can run instead. And even if a process never blocks, the Erlang runtime is free to switch execution from that process to another at any time. In other words, Erlang has a non-cooperative concurrency model.

Notice that `get_poetry/4`, after receiving a bit of poem, proceeds by recursively calling itself. To an imperative language programmer this might seem like a recipe for running out of memory, but the Erlang compiler can optimize "tail" calls (function calls that are the last statement in a function) into loops. And this highlights another curious parallel between the Erlang and Twisted clients. In the Twisted client, the "virtual" loops are created by the reactor calling the same function (`dataReceived`) over and over again. And in the Erlang client, the "real" processes running (`get_poetry/4`) form loops by calling *themselves* over and over again via tail-call optimization. How about that.

If the connection is closed, the last thing `get_poetry` does is send the poem to the main process. That also ends the process that `get_poetry` is running, as there is nothing left for it to do.

The remaining key function in our Erlang client is `collect_poems`:

```
1  collect_poems(0, Poems) ->
2      [io:format("~s\n", [P]) || P <- Poems];
3  collect_poems(N, Poems) ->
4      receive
5          {'DOWN', _, _, _, _} ->
6              collect_poems(N-1, Poems);
7          {poem, Poem} ->
8              collect_poems(N, [Poem|Poems])
9      end.
```

This function is run by the main process and, like `get_poetry`, it recursively loops on itself. It also blocks. The `receive` statement tells the process to wait for a message to arrive that matches one of the given patterns, and
then extract the message from its "mailbox".

The `collect_poems` function waits for two kinds of messages: poems and "DOWN" notifications. The latter is a message sent to the main process when one of the `get_poetry` processes dies for any reason (this is the `monitor` part of `spawn_monitor`). By counting `DOWN` messages, we know when all the poetry has finished. The former is a message from one of the `get_poetry` processes containing one complete poem.

Ok, let's take the Erlang client out for a spin. First start up three slow poetry servers:

```
python blocking-server/slowpoetry.py --port 10001 poetry/fascination.txt
python blocking-server/slowpoetry.py --port 10002 poetry/science.txt
python blocking-server/slowpoetry.py --port 10003 poetry/ecstasy.txt --num-bytes 30
```

Now we can run the Erlang client, which has a similar command-line syntax as the Python clients. If you are on a Linux or other UNIX-like system, then you should be able to run the client directly (assuming you have Erlang installed and available in your `PATH`). On Windows you will probably need to run the `escript` program, with the path to the Erlang client as the first argument (with the remaining arguments for the Erlang client itself).

```
./erlang-client-1/get-poetry 10001 10002 10003
```

After that you should see output like this:

```
Task 3: got 30 bytes of poetry from 127:0:0:1:10003
Task 2: got 10 bytes of poetry from 127:0:0:1:10002
Task 1: got 10 bytes of poetry from 127:0:0:1:10001
...
```

This is just like one of our earlier Python clients where we print a message for each little bit of poetry we get. When all the poems have finished the client should print out the complete text of each one. Notice the client is switching back and forth between all the servers depending on which one has some poetry to send.

Figure 45 shows the process structure of our Erlang client:

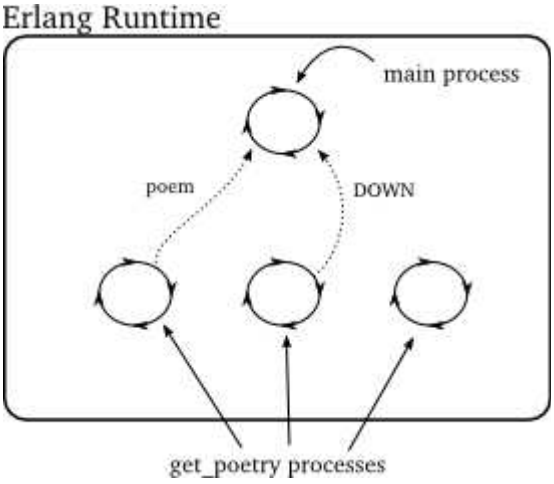

Figure 45: Erlang poetry client

This figure shows three `get_poetry` processes (one per server) and one main process. You can also see the messages that flow from the poetry processes to main process.

So what happens if one of those servers is down? Let's try it:

```
./erlang-client-1/get-poetry 10001 10005
```

The above command contains one active port (assuming you left all the earlier poetry servers running) and one inactive port (assuming you aren't running any server on port 10005). And we get some output like this:

```
Task 1: got 10 bytes of poetry from 127:0:0:1:10001

=ERROR REPORT==== 25-Sep-2010::21:02:10 ===
Error in process <0.33.0> with exit value: {{badmatch,{error,econnrefused}},[{erl_eval
```

```
Task 1: got 10 bytes of poetry from 127:0:0:1:10001
Task 1: got 10 bytes of poetry from 127:0:0:1:10001
...
```

And eventually the client finishes downloading the poem from the active server, prints out the poem, and exits. So how did the `main` function know that both processes were done? That error message is the clue. The error happens when `get_poetry` tries to connect to the server and gets a connection refused error instead of the expected value (`{ok, Socket}`). The resulting exception is called `badmatch` because Erlang "assignment" statements are really pattern-matching operations.

An unhandled exception in an Erlang process causes the process to "crash", which means the process stops running and all of its resources are garbage collected. But the `main` process, which is monitoring all of the `get_poetry` processes, will receive a `DOWN` message when any of those processes stops running for any reason. And thus our client exits when it should instead of running forever.

Discussion

Let's take stock of some of the parallels between the Twisted and Erlang clients:

1. Both clients connect (or try to connect) to all the poetry servers at once.
2. Both clients receive data from the servers as soon as it comes in, regardless of which server delivers the data.
3. Both clients process the poetry in little bits, and thus have to save the portion of the poems received thus far.
4. Both clients create an "object" (either a Python object or an Erlang process) to handle all the work for one particular server.
5. Both clients have to carefully determine when all the poetry has finished, regardless of whether a particular download succeeded or failed.

And finally, the `main` functions in both clients asynchronously receive poems and "task done" notifications. In the Twisted client this information is delivered via a `Deferred` while the Erlang client receives inter-process messages.

Notice how similar both clients are, in both their overall strategy and the structure of their code. The mechanics are a bit different, with objects, deferreds, and callbacks on the one hand and processes and messages on the other. But the high-level mental models of both clients are quite similar, and it's pretty easy to move from one to the other once you are familiar with both.

Even the reactor pattern reappears in the Erlang client in miniaturized form. Each Erlang process in our poetry client eventually turns into a recursive loop that:

1. Waits for something to happen (a bit of poetry comes in, a poem is delivered, another process finishes), and
2. Takes some appropriate action.

You can think of an Erlang program as a big collection of little reactors, each spinning around and occasionally sending a message to another little reactor (which will process that message as just another event).

And if you delve deeper into Erlang you will find callbacks making an appearance. The Erlang `gen_server` process is a generic reactor loop that you "instantiate" by providing a fixed set of callback functions, a pattern repeated elsewhere in the Erlang system.

So if, having learned Twisted, you ever decide to give Erlang a try I think you will find yourself in familiar mental territory.

Further Reading

In this Part we've focused on the similarities between Twisted and Erlang, but there are of course many differences. One particularly unique feature of Erlang is its approach to error handling. A large Erlang program is structured as a tree of processes, with "supervisors" in the higher branches and "workers" in the leaves. And if a worker process crashes, a supervisor process will notice and take some action (typically restarting the failed worker).

If you are interested in learning more Erlang then you are in luck. Several Erlang books have either been published recently, or will be published shortly:

- Programming Erlang — written by one of Erlang's inventors. A great introduction to the language.
- Erlang Programming — this complements the Armstrong book and goes into more detail in several key areas.
- Erlang and OTP in Action — this hasn't been published yet, but I am eagerly awaiting my copy. Neither of the first two books really addresses OTP, the Erlang framework for building large apps. Full disclosure: two of the authors are friends of mine.

Well that's it for Erlang. In the next Part we will look at Haskell, another functional language with a very different feel from either Python or Erlang. Nevertheless, we shall endeavor to find some common ground.

Suggested Exercises for the Highly Motivated

1. Go through the Erlang and Python clients and identify where they are similar and where they differ. How do they each handle errors (like a failure to connect to a poetry server)?
2. Simplify the Erlang client so it no longer prints out each bit of poetry that comes in (so you don't need to keep track of task numbers either).
3. Modify the Erlang client to measure the time it takes to download each poem.
4. Modify the Erlang client to print out the poems in the same order as they were given on the command line.
5. Modify the Erlang client to print out a more readable error message when we can't connect to a poetry server.
6. Write Erlang versions of the poetry servers we made with Twisted.

Part 21: Lazy is as Lazy Doesn't: Twisted and Haskell

This continues the introduction started here. You can find an index to the entire series here.

Introduction

In the last Part we compared Twisted with Erlang, giving most of our attention to some ideas they have in common. And that ended up being pretty simple, as asynchronous I/O and reactive programming are key components of the Erlang runtime and process model.

Today we are going to range further afield and look at Haskell, another functional language that is nevertheless quite different from Erlang (and, of course, Python). There won't be as many parallels, but we will nevertheless find some asynchronous I/O hiding under the covers.

Functional with a Capital F

Although Erlang is also a functional language, its main focus is a reliable concurrency model. Haskell, on the other hand, is functional through and through, making unabashed use of concepts from category theory like functors and monads.

Don't worry, we're not going into any of that here (as if we could). Instead we'll focus on one of Haskell's more traditionally functional features: laziness. Like many functional languages (but unlike Erlang), Haskell supports lazy evaluation. In a lazily evaluated language the text of a program doesn't so much describe how to compute something as what to compute. The details of actually performing the computation are generally left to the compiler and runtime system.

And, more to the point, as a lazily-evaluated computation proceeds the runtime may evaluate expressions only partially (lazily) instead of all at once. In general, the runtime will evaluate only as much of an expression as is needed to make progress on the current computation.

Here is a simple Haskell statement applying `head`, a function that retrieves the first element of a list, to the list `[1,2,3]` (Haskell and Python share some of their list syntax):

```
  head [1,2,3]
```

If you install the GHC Haskell runtime, you can try this out yourself:

```
[~] ghci
GHCi, version 6.12.1: http://www.haskell.org/ghc/  : ? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> head [1,2,3]
1
Prelude>
```

The result is the number 1, as expected.

The Haskell list syntax includes the handy ability to define a list from its first couple of elements. For example, the list `[2,4 ..]` is the sequence of even numbers starting with 2. Where does it end? Well, it doesn't. The Haskell list [2,4 ..] and others like it represent (conceptually) infinite lists. You can see this if you try to evaluate one at the interactive Haskell prompt, which will attempt to print out the result of your expression:

```
Prelude> [2,4 ..]
[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50,52,54,56,58,60
...
```

You'll have to press `Ctrl-C` to stop that computation as it will never actually terminate. But because of lazy evaluation, it is possible to use these infinite lists in Haskell with no trouble:

```
Prelude> head [2,4 ..]
2
Prelude> head (tail [2,4 ..])
4
Prelude> head (tail (tail [2,4 ..]))
6
```

Here we are accessing the first, second, and third elements of this infinite list respectively, with no infinite loop anywhere in sight. This is the essence of lazy evaluation. Instead of first evaluating the entire list (which would cause an infinite loop) and then giving that list to the `head` function, the Haskell runtime only constructs as much of the list as it needs for `head` to finish its work. The rest of the list is never constructed at all, because it is not needed to proceed with the computation.

When we bring the `tail` function into play, Haskell is forced to construct the list further, but again only as much as it needs to evaluate the next step of the computation. And once the computation is done, the (unfinished) list can be discarded.

Here's some Haskell code that partially consumes three different infinite lists:

```
Prelude> let x = [1..]
```

```
Prelude> let y = [2,4 ..]
Prelude> let z = [3,6 ..]
Prelude> head (tail (tail (zip3 x y z)))
(3,6,9)
```

Here we zip all the lists together, then grab the head of the tail of the tail. Once again, Haskell has no trouble with this and only constructs as much of each list as it needs to finish evaluating our code. We can visualize the Haskell runtime "consuming" these infinite lists in Figure 46:

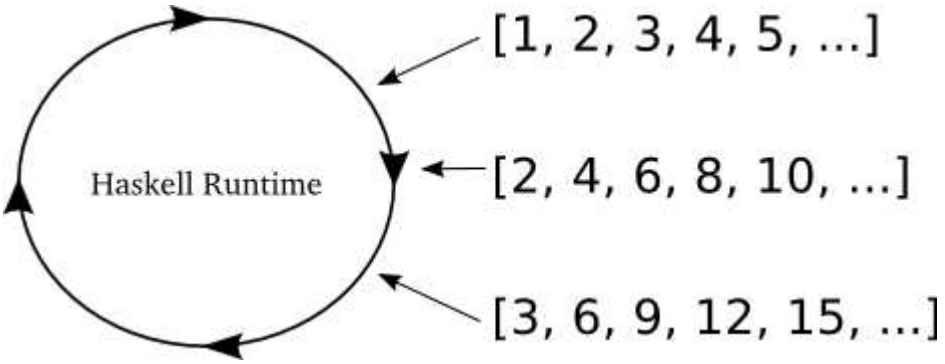

Figure 46: Haskell consuming some infinite lists

Although we've drawn the Haskell runtime as a simple loop, it might be implemented with multiple threads (and probably is if you are using the GHC version of Haskell). But the main point to notice is how this figure looks like a reactor loop consuming bits of data as they come in on network sockets.

You can think of asynchronous I/O and the reactor pattern as a very limited form of lazy evaluation. The asynchronous I/O motto is: "Only process as much data as you have". And the lazy evaluation motto is: "Only process as much data as you need". Furthermore, a lazily-evaluated language applies that motto almost everywhere, not just in the limited scope of I/O.

But the point is that, for a lazily-evaluated language, making use of asynchronous I/O is no big deal. The compiler and runtime are already designed to process data structures bit by bit, so lazily processing the incoming chunks of an I/O stream is just par for the course. And thus the Haskell runtime, like the Erlang runtime, simply incorporates asynchronous I/O as part of its socket abstractions. And we can show that by implementing a poetry client in Haskell.

Haskell Poetry

Our first Haskell poetry client is located in `haskell-client-1/get-poetry.hs`. As with Erlang, we're going to jump straight to a finished client, and then suggest further reading if you'd like to learn more.

Haskell also supports light-weight threads or processes, though they aren't as central to Haskell as they are to Erlang, and our Haskell client creates one process for each poem we want to download. The key function there is `runTask` which connects to a socket and starts the `getPoetry` function in a light-weight thread.

You'll notice a lot of type declarations in this code. Haskell, unlike Python or Erlang, is statically typed. We don't declare types for each and every variable because Haskell will automatically infer types not explicitly declared (or report an error if it can't). A number of the functions include the `IO` type (technically a monad) because Haskell requires us to cleanly separate code with side-effects (i.e., code that performs I/O) from pure functions.

The `getPoetry` function includes this line:

```
poem <- hGetContents h
```

which appears to be reading the entire poem from the handle (i.e., the TCP socket) at once. But Haskell, as usual, is lazy. And the Haskell runtime includes one or more actual threads which perform asynchronous I/O in a `select` loop, thus preserving the possibilities for lazy evaluation of I/O streams.

Just to illustrate that asynchronous I/O is really going on, we have included a "callback" function, `gotLine`, that prints out some task information for each line in the poem. But it's not really a callback function at all, and the program would use asynchronous I/O whether we included it or not. Even calling it "gotLine" reflects an imperative-language mindset that is out of place in a Haskell program. No matter, we'll clean it up in a bit, but let's take our first Haskell client out for a spin. Start up some slow poetry servers:

```
python blocking-server/slowpoetry.py --port 10001 poetry/fascination.txt
python blocking-server/slowpoetry.py --port 10002 poetry/science.txt
python blocking-server/slowpoetry.py --port 10003 poetry/ecstasy.txt --num-bytes 30
```

Now compile the Haskell client:

```
cd haskell-client-1/
ghc --make get-poetry.hs
```

This will create a binary called `get-poetry`. Finally, run the client against our servers:

```
./get-poetry 10001 10002 1000
```

And you should see some output like this:

```
Task 3: got 12 bytes of poetry from localhost:10003
Task 3: got 1 bytes of poetry from localhost:10003
Task 3: got 30 bytes of poetry from localhost:10003
Task 2: got 20 bytes of poetry from localhost:10002
Task 3: got 44 bytes of poetry from localhost:10003
Task 2: got 1 bytes of poetry from localhost:10002
Task 3: got 29 bytes of poetry from localhost:10003
Task 1: got 36 bytes of poetry from localhost:10001
Task 1: got 1 bytes of poetry from localhost:10001
...
```

The output is slightly different than previous asynchronous clients because we are printing one line for each line of poetry instead of each arbitrary chunk of data. But, as you can see, the client is clearly processing data from all the servers together, instead of one after the other. You'll also notice that the client prints out the first poem as soon as it's finished, without waiting for the others, which continue on at their own pace.

Alright, let's clean the remaining bits of imperative cruft from our client and present a version which just grabs the poetry without bothering with task numbers. You can find it in `haskell-client-2/get-poetry.hs`. Notice that it's much shorter and, for each server, just connects to the socket, grabs all the data, and sends it back.

Ok, let's compile a new client:

```
cd haskell-client-2/
ghc --make get-poetry.hs
```

And run it against the same set of poetry servers:

```
./get-poetry 10001 10002 10003
```

And you should see the text of each poem appear, eventually, on the screen.

You will notice from the server output that each server is sending data to the client simultaneously. What's more, the client prints out each line of the first poem as soon as possible, without waiting for the

rest of the poem, even while it's working on the other two. And then it quickly prints out the second poem, which it has been accumulating all along.

And all of that happens without us having to do much of anything. There are no callbacks, no messages being passed back and forth, just a concise description of what we want the program to do, and very little in the way of how it should go about doing it. The rest is taken care of by the Haskell compiler and runtime. Nifty.

Discussion and Further Reading

In moving from Twisted to Erlang to Haskell we can see a parallel movement, from the foreground to the background, of the ideas behind asynchronous programming. In Twisted, asynchronous programming is the central motivating idea behind Twisted's existence. And Twisted's implementation as a framework separate from Python (and Python's lack of core asynchronous abstractions like lightweight threads) keeps the asynchronous model front and center when you write programs using Twisted.

In Erlang, asynchronicity is still very visible to the programmer, but the details are now part of the fabric of the language and runtime system, enabling an abstraction in which asynchronous messages are exchanged between synchronous processes.

And finally, in Haskell, asynchronous I/O is just another technique inside the runtime, largely unseen by the programmer, for providing the lazy evaluation that is one of Haskell's central ideas.

We don't have any profound insight into this situation, we're just pointing out the many and interesting places where the asynchronous model shows up, and the many different ways it can be expressed.

And if any of this has piqued your interest in Haskell, then we can recommend Real World Haskell to continue your studies. The book is a model of what a good language introduction should be. And while I haven't read it, I've heard good things about Learn You a Haskell.

This brings us to the end of our tour of asynchronous systems outside of Twisted, and the penultimate part in our series. In Part 22 we will conclude, and suggest ways to learn more about Twisted.

Suggested Exercises for the Startlingly Motivated

1. Compare the Twisted, Erlang, and Haskell clients with each other.
2. Modify the Haskell clients to handle failures to connect to a poetry server so they download all the poetry they can and output reasonable error messages for the poems they can't.
3. Write Haskell versions of the poetry servers we made with Twisted.

Part 22: The End

This concludes the introduction started here. You can find an index to the entire series here.

All Done

Whew! Thank you for sticking with me. When I started this series I didn't realize it was going to be this long, or take this much time to make. But I have enjoyed creating it and hope you have enjoyed reading it.

Now that I have finished, I will look into the possibility of generating a PDF format. No promises, though.

I would like to conclude with a few suggestions on how to continue your Twisted education.

Further Reading

First, I would recommend reading the Twisted online documentation. Although it is much-maligned, I think it's better than it is often given credit for.

If you want to use Twisted for web programming, then Jean-Paul Calderone has a well-regarded series called "Twisted Web in 60 Seconds". I suspect it will take a little longer than that to read, though.

There is also a Twisted Book, which I can't say much about as I haven't read it.

But more important than any of those, I think, is to read the Twisted source code. Since that code is written by people who know Twisted very well, it is an excellent source of examples for how do to things the "Twisted Way".

Suggested Exercises

1. Port a synchronous program you wrote to Twisted.
2. Write a new Twisted program from scratch.
3. Pick a bug from the Twisted bug database and fix it. Submit a patch to the Twisted developers. Don't forget to read about the process for making your contribution.

The End, Really

Happy Hacking!

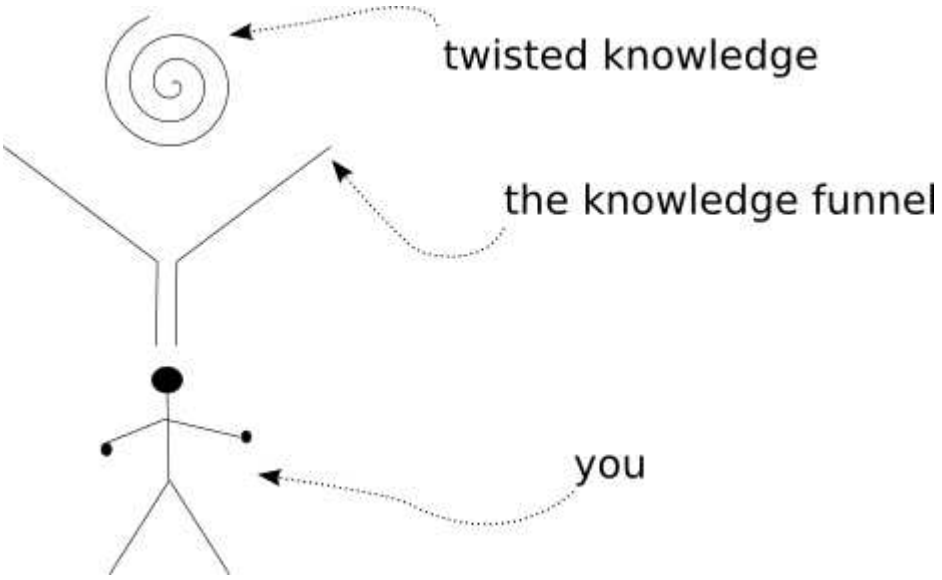

Figure 47: The End