Dispatcher

Role:

Serve as an interface between front end and scheduler

* Receive queries from front end
* Receive updates from scheduler
* Maintain a Q of most able ‘worker’ and send out Q head whenever a request is made

Coded in Python Twisted

Event-driven networking engine written in Python and licensed under the MIT license.

Why twisted:

* Flexibility: codebase is small and it contains many hooks for dynamic content. Can run both on Linux and Windows
* Available source code and an open license
* Security: Twisted is written in Python, a high-level language, rendering it immune to the most common class of security flaw in network software, the "buffer overflow" which have allowed worms and crackers to tamper with or disable internet servers. Also takes advantage of operating-system security features whenever possible, delegating responsibility for tasks to the appropriate user to make the system more difficult to exploit.
* Stability: Thanks to Python's error handling mechanisms, the Twisted server framework is extremely stable



<http://twistedmatrix.com/projects/core/documentation/howto/overview.html>

Introduction to concurrent programming

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

 1. it is computationally intensive (for example factorising large numbers) and requires a certain amount of CPU time to calculate the answer; or

 2. it is not computationally intensive but has to wait for data to be available to produce a result.

Waiting for answers

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

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

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

Not waiting on data

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

 1. handle each connection in a separate operating system process, in which case the operating system will take care of letting other processes run while one is waiting;

 2. handle each connection in a separate thread1 in which the threading framework takes care of letting other threads run while one is waiting; or

 3. use non-blocking system calls to handle all connections in one thread.

Non-blocking calls

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

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

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

 1. it calls a connection function to connect to the remote server;

 2. the connection function returns immediately, with the implication that the notify the email sending library will be called when the connect has been made; and

 3. once the connection is made, the connect mechanism notifies the email sending function that the connection is ready.

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

Callbacks

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

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

Deferreds

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

The Problem that Deferreds Solve

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

Deferreds are designed to enable Twisted programs to wait for data without hanging until that data arrives. They do this by giving a simple management interface for callbacks to libraries and applications. Libraries know that they always make their results available by calling Deferred.callback and errors by calling Deferred.errback. Applications set up result handlers by attaching callbacks and errbacks to deferreds in the order they want them called.

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

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

Deferreds - a signal that data is yet to come

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

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

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

Callbacks

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

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

Separation of protocols and transports

Twisted is designed for complete separation between logical protocols (usually relying on stream-based connection semantics, such as HTTP or POP3) and physical transport layers supporting such stream-based semantics (such as files, sockets or SSL libraries). Connection between a logical protocol and a transport layer happens at the last possible moment—just before information is passed into the logical protocol instance. The logical protocol is informed of the transport layer instance, and can use it to send messages back and to check for the peer's identity. Note that it is still possible, in protocol code, to deeply query the transport layer on transport issues (such as checking a client-side SSL certificate). Naturally, such protocol code will fail (raise an exception) if the transport layer does not support such semantics.

[edit] Deferreds

Central to the Twisted application model is the concept of a deferred (elsewhere called a future). A deferred is a value which has not been computed yet, for example because it needs data from a remote peer. Deferreds can be passed around, just like regular objects, but cannot be asked for their value. Each deferred supports a callback chain. When the deferred gets the value, it is transferred through the callback chain, with the result of each callback being the input for the next one. This allows operating on the values of a deferred without knowing what they are. For example, if a deferred returns a string from a remote peer containing an IP address in quad format, a callback can be attached to translate it into a 32-bit number. Any user of the deferred can now treat it is as a deferred returning a 32-bit number. This, and the related ability to define "errbacks" (callbacks which are called as error handlers), allows code which looks as though it is serial, while still maintaining the event-driven abstraction.

[edit] Thread support

Twisted supports an abstraction over raw threads—using a thread as a deferred source. Thus, a deferred is returned immediately, which will receive a value when the thread finishes. Callbacks can be attached which will run in the main thread, thus alleviating the need for complex locking solutions. A prime example of such usage, which comes from Twisted's support libraries, is using this model to call into databases. The database call itself happens on a foreign thread, but the analysis of the result happens in the main thread.

[edit] Foreign loop support

Twisted can integrate with foreign event loops, such as those of GTK+, Qt and Cocoa (through PyObjC). This allows using Twisted as the networking support layer in GUI programs, using all of its libraries without adding a thread-per-socket overhead, as using Python's native library would. A full-fledged web server can be integrated in-process with a GUI program using this model, for example.