PROGRAMMING, SYSTEMS, LEADERSHIP AND /RANDOM

FEBRUARY 11, 2009

TWISTED - HELLO, ASYNCHRONOUS PROGRAMMING

by jesse in Programming, pycon 2009, Python

Note: This is the third post in what I hope will be a series leading up to my concurrency/distributed systems talk at PyCon. I'm steadily working through experimenting with and learning the various frameworks/libraries in the python ecosystem.

I reserve the right (and probably will) to revise these entries based on feedback from people (mainly the author(s) of said tool(s)). I will also add additional bits and pieces as I learn and explore more. Additionally, thanks to glyph for giving me a hell of a lot of feedback.**/Note**

Twisted is the 800 lbs gorilla of the "concurrency" frameworks. It's been around for awhile, has a large following - it's used by everyone from Apple (iCal server) to Buildbot Buildbot. It has a literal ton of sub projects and other "semi attached appendages". Twisted can be daunting for almost everyone - while it is conceptually simple, the docs and examples could be more approachable. People look at Twisted as an all-or-nothing bet when considering it for their applications, which to an extent, it is.

But if I am going to do a concurrency/distributed systems talk - I can't ignore one of the most widely used and original frameworks in this space.

So, as always - these are my semi-rough notes diving into Twisted-core. Like Kamaelia, I am going to side step the asteroid ring which surrounds Twisted (or tentacles... I can't decide which to use) and delve into the core.

Moving along - Twisted is based around asynchronous programming - a model for adding a great deal of concurrency to your application

via non blocking calls or isolating blocking calls "elsewhere". This is largely the same approach that GUI toolkits use wherein a given event is assigned something to run when an "event" occurs, such as a button click or data is available. Glyph sent me a very simple side-by-side:

```
reactor.listenTCP(8080, someFactory)

button.connect('clicked', someCallback)
```

This shows the essential aspect of asynchronus/event-driven systems - you tell x that when y occurs, call z. It really is conceptually simple. The main loop of the application simply focuses on constructing these relationships, and executing the callbacks when the event occurs.

Twisted's focus is on network-based applications - these mesh well with the idea of non-blocking I/O, where you have to fundamentally chunk your work up into small pieces which take a very short time to execute. Networked apps spend most of their time waiting for data to come in over the wire. Twisted also has faculties to isolate CPU intensive and/or blocking calls within threads or processes.

I'm going to focus on two of the core components - Deferreds and the Reactor, this should help illustrate what the core paradigm is.

Deferreds are a core component of Twisted - a deferred in the simplest terms is an object that when created, represents some Thing which will eventually return Something or Error - a placeholder for something in the future. The way you handle Something or Error is you tell the deferred that if it gets Something, it should call consume_function, otherwise - if it gets an Error, it should call error_function. Easy!

Take the below for example - read_mail and error_function are what's known as callbacks - a function which is called by something else when something occurs (I hate me for writing that).

Callbacks are really simple, as illustrated here:

```
import os, sys
def read_mail(mailitems):
    print mailitems
    sys.exit()

def read_error(error):
    raise Exception('error: %s' % error)

def wait_for_mail(callback, errback):
    while True:
        try:
            mail = os.path.isfile('mail')
```

```
            if mail:
                callback(open('mail','r').readlines())
            else:
                pass
        except Exception, e:
            errback(e)

wait_for_mail(read_mail, read_error)
```

Sidenote: Callbacks are stupid easy with python. I love them. Heck, you can pickle a callback and shoot it over the wire to another machine. Callbacks are cool. You should watch Alex Martelli's callback talk on youtube.

In any case, you tell a deferred (the object which represents a promise of something) what to do when data is returned - you do this by generating a deferred, and then adding callbacks onto it - note that the function wait_for_mail needs to **return** a deferred. In this toy example, I want to just look for a "mail file" on disk, and then if it exists, return a string to the callback:

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

def read_mail(mailitems):
    print mailitems
    reactor.stop()

def wait_for_mail(d=None):
    if not d:
        d = defer.Deferred()
    if not os.path.isfile('mail'):
        reactor.callLater(1, wait_for_mail, d)
    else:
        d.callback(open('mail','r').readlines())
    return d

deferred = wait_for_mail()
deferred.addCallback(read_mail)
reactor.callLater(60, reactor.stop)
reactor.run()
```

I wanted to keep this as simple as possible, as it illustrates some things that fundamentally tripped me up at first.

First typically, if you were to solve a problem - say, polling a mailbox, you might do this:

```
import os, sys, threading, time
def read_mail(mailitems):
    print mailitems
```

```
def wait_for_mail(reader):
   while not os.path.isfile('mail'):
      time.sleep(.1)

   reader(open('mail','r').readlines())

t = threading.Thread(target=wait_for_mail, args=(read_mail,))
t.start()
t.join()
sys.exit()
```

Or some other pattern of spawning a thread and then waiting for that thread to chuck the data back to you. The thread gets out of your way, and doesn't force the main part of the program to block, sort of. In fact, with Twisted, the main part of your application **is required** not to block.

As a point of order, you could do the same thing with Twisted like this:

```
...
d = deferToThread(wait_for_mail)
...
```

Threads are the common way of pushing blocking work off to the side - most of us have had to do it at one point or another. In the Twisted world, this has to be turned on it's head a bit.

In Twisted, you have to split your problem into small, individual functions/methods - ideally, you isolate the really blocking part (say, waiting for a file to appear) in it's very own function. You make the non blocking parts - say checking initial existence and constructing the deferred object run as quickly as possible, and then return the deferred - a promise of data to come via the slow method. The slow part in the threading example is the time.sleep().

You then **schedule** that blocking call to run, it shouldn't block, but rather it should poll for data or changes in it's buffer(s) and either reschedule itself to run if there is no data, or return the data if there is. This event is time based - but the same applies to adding a callback to a non-time-based item, such as setting up something which listen on a socket for data.

The fact you schedule work within the reactor tripped me up at first. I was thinking in blocking terms though (Twisted has a faculty for passing blocking work off to threads via the deferToThread call) - the function kept wanting to block instead of polling, or looking for state change. Everything needs to be scheduled in one way or another.

Glyph wisely pointed out that this is a common issue with people rethinking concurrency in terms of "discrete events". For example,

most people are content to think about concurrency in terms of workers "who are off doing things". In theory, those workers are "always doing something" - in reality, the operating system is simply suspending your worker(s) until an interrupt (i.e.) discreet event occurs, which causes the worker(s)/app unblock.

Glyph suggest the following as a good example using generator syntax:

```
from twisted.internet.defer import inlineCallbacks, returnValue
from twisted.internet.task import deferLater
from twisted.internet import reactor

def deferredSleep(howLong):
   return deferLater(reactor, howLong, lambda : None)

@inlineCallbacks
def wait_for_mail():
   while True:
      if os.path.isfile('mail'):
         returnValue(open('mail','r').readlines())
      yield deferredSleep(1.0)

@inlineCallbacks
def check_mail():
   mail = yield wait_for_mail()
   print 'got mail', mail
```

The example he provided is interesting - it has no "scheduling" involved, and instead it uses something I didn't see originally - deferLater and inlineCallback, inlineCallback accepts a function as an argument, that function can yield a deffered or call returnValue, essentially anywhere where you would normally block, you simply yield. In this case, we simply call deferLater if the file doesn't exist, which tells the reactor to re-run this some time in the future.

Before I move off the basic view of deferreds - there's something to note, deferreds can accept a chain of callbacks:

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

def read_mail(mailitems):
   print mailitems
   return "this %s is junk" % mailitems

def shred_mail(mailitems):
   print 'buzzzzz: %s' % mailitems
   reactor.stop()

def wait_for_mail(d=None):
```

```
    if not d:
        d = defer.Deferred()
    if not os.path.isfile('mail'):
        reactor.callLater(1, wait_for_mail, d)
    else:
        d.callback('letter')
    return d

deferred = wait_for_mail()
```



```
)
)
```

buzzzzz: this letter is junk

This allows each callback chained onto a deferred to alter the data in some form - the modified data is passed to the next callback in the chain. It's not really magic. It's simply a series of functions to call when an event occurs.

One thing to keep in mind when thinking about Twisted - Twisted is **single threaded**. Ok, not really - sort of. Glyph called me out on this, and rightly so. In reality all I/O in Twisted is single-threaded, most Twisted APIs are not thread safe and deferred callback chains execute in a single thread... But - Twisted does support both threads and processes. There is no reason why you can't use a library which uses threads in your twisted application for example. You can use threads with twisted, so don't worry too much about that.

On the other hand, if your entire application is a thread-spawnfest, you might want to reconsider - the design of your app, that is. /flamebait

All code executes in the main thread of a single python process, which is why when discussing deferreds and blocking calls, it is so important to break your problem down into the smallest steps possible and isolate/rework blocking code into deferred actions.

Onto the reactor then.

The reactor is the event loop mechanism for Twisted. It takes care of executing all of the various timed actions and the execution of the callback/errback stack. Timed actions can be deferreds, etc. Deferreds are simply objects executed by the Reactor.

You'll notice in the example above, we didn't create an instance of the reactor, instead we just imported it. If you look at Twisted.internet.reactor - you'll see this removes any previous instances of reactor in sys.modules and then calls install() on the

target reactor. This means the reactor is a singleton, all future imports/calls will always refer to this reactor.

Now, the documentation in internet.reactor mentions that new application code should pass around an instance of a reactor - this moves away from the reactor-as-a-singleton (the simple behavior) and into something a bit more interesting. Glyph pointed out the obvious usefulness for this - testability, you can pass in a reactor rather than grabbing it from the global, you have more control.

You can also use this to group a series of actions both timed and otherwise, connections/etc within a given reactor, and then create a meta-reactor to control the multiple reactors. Reactors, all the way down.

See, Twisted has multiple types of reactors - there are reactors based on select (the default), GTK, Cocoa (PyObjc), etc. Each reactor manages the scheduling and execution of the tasks added to it in it's on unique style, but implement a common interface. Application code should not **care** what reactor is running - the reactor is an abstraction above other, some operating system optimized polling/looping mechanisms. For example, GTK+ looping and polling, select on Linux/etc.

Quoting Glyph:

> The idea of all this is that Twisted code is at the top of the food chain. GTK+ networking code can only run in GTK+ programs, kqueue networking code can only run on FreeBSD machines, but Twisted networking code can run anywhere it can get its hands on something that looks even vaguely like select(). If you want a Windows desktop program and a UNIX server program to run the same networking code, but have radically different event APIs under the covers for good performance, Twisted has you covered.

You can install a given reactor by doing the from twisted.xxx.reactor_name and calling the install() method (which is all twisted.internet.reactor does). The preferred method is to use the "--reactor" argument to the twistd/trial tools.

Twisted is fundamentally a networking stack: it's built to solve non computationally intensive tasks that require a lot of waiting and polling for data: networking fits perfectly into this. With additions - it can also serve as a platform for computationally intensive/distributed applications. For example, the ampoule add on.

In the networking sense: Twisted is perfect as long as you can deconstruct and rethink the way you solve day to day problems, remove or rethink blocking code, switch your application model to

something event/message driven.

Admittedly; Twisted isn't for me (right now) - but with time, it could be, and it could work great for your application today. It has libraries for just about any protocol you could possibly ask for. It's code base is **huge** and actually has some pretty cool code inside of it. The catch is - you don't port an application **to** Twisted: You write a Twisted application in most cases.

However; it can also be hard to approach - both within the code, and the documentation. You see questions pop up all the time "I think Twisted does this" - and while it probably does it could take you awhile just to grok what it is to be Twisted.

For example - dumb down the examples. While the finger tutorial is "the introduction" starting down on a level where you take a normal, single threaded application (perhaps using generators), port it to threads, point out the issues there, and then port it to twisted, using as little "twisted magic" as possible.

This isn't to say it's impossible to grok; but helping walk people through learning how to begin to think asynchronously, rather than explaining the esoteric or the One True way of doing something, consider the positive feedback that Steve Holden's "Teach Me Twisted" got (summary here). Assume most people can't spell asynchronous, and build it up.

The Django documentation is probably one of my favorite examples of this - it starts very simple and very approachable and walks the user through everything "from the beginning".

RELATED LINKS

- http://stackoverflow.com/questions/80617/asychronous-programming-in-python-twisted/81456#81456
- http://www.usenix.org/events/usenix03/tech/freenix03/full_papers/lefkowitz/lefkowitz_html/ind
- http://mumak.net/stuff/Twisted-intro.html
- http://sluggo.scrapping.cc/python/Twisted_finger_gentle.txt
- http://Twistedmatrix.com/projects/core/documentation/howto/defer.html
- http://Twistedmatrix.com/projects/core/documentation/howto/gendefer.html
- http://Twistedmatrix.com/projects/core/documentation/howto/async.html
- http://Twistedmatrix.com/projects/core/documentation/howto/reactor-basics.html
- http://www.nightmare.com/pythonwin/async_sockets.html
- http://enthusiasm.cozy.org/archives/2009/02/twisted-nevermind

‹ PREVIOUS POST
PYCON 2009: IN UR ...

NEXT POST
SSH PROGRAMMING WITH ... ›