Keith Rarick

Asynchronous Programming in Python

10 DEC 2009

Twisted is pretty good. It sits as one of the top networking libraries in Python, and with good reason. It is properly asynchronous, flexible, and mature. But it also has some pretty serious flaws that make it harder than necessary for programmers to use.

This hinders adoption of Twisted, and (worse) it hinders adoption of asynchronous programming in general. Unfortunately, fixing most of these flaws in the context of Twisted would cause massive compatibility problems. This makes me think the world could use a new, Pythonic, asynchronous programming library. Perhaps it should be a fork of Twisted; perhaps it should be a brand-new project. Either way, it would make life much nicer for programmers like you and me in the future.

TOWARD A BETTER EVENT LIBRARY

Here is what Twisted gets right:

- Pervasive asynchrony.
- The “reactor” pattern.
- Each asynchronous function does not take callbacks as parameters; it just returns a promise (which Twisted calls a “deferred”).
- Able to use modern select()-replacements like kqueue, epoll, etc.
- Integrates with the GTK+ event loop. Same for other GUI toolkits.

These things are all absolutely crucial and Twisted nails them. This is what makes Twisted so great.

Here is what I would do differently:

- *Limited scope* – This project has no need to include dozens of incomplete and neglected protocols. Instead, such things can easily (and should) be maintained as separate projects. For example, we don't need two-and-a-half half-assed HTTP modules, but what if we had one excellent asynchronous HTTP library, which incidentally achieves asynchrony by means of this event library. It might start off as a port of httplib2, which is well designed even if it lacks asynchrony. This would leave the maintainers of the core event system more time to focus on making a really coherent, "as-simple-as-possible-but-no-simpler", useful tool.

- *User-focused design* – A good library, like a good language, should make simple things simple and hard things possible. Twisted makes simple things possible and hard things possible. Most users don't particularly want to extend base classes to implement derived Factories and Protocols and Clients, they just want to fetch the document at a URL (asynchronously!). Achieving this ease of use is not terribly hard, but it requires conscious effort. You must start by designing the ideal top-level interface, then work downward and make it operate correctly. Twisted's web modules (I don't mean to pick only on HTTP here; these are just examples) look to me as if they started from the basic building blocks and added on pieces until HTTP was achieved. We are left with whatever interface happened to come out at the end. Further, they look like they were written by former C++ and Java programmers who haven't yet fully realized that Python code doesn't have to be so complicated.

- *Arbitrary events* – Promises should let you send more than just "success" and "failure". You should be able to emit arbitrarily-named events. For example, suppose you make an HTTP request. In the simple case, you just want the complete document when it is fully received. But what if you also want to

update a progress bar for the transfer? You shouldn't have to start digging through the HTTP library's low-level unbuffered innards. Instead, the promise that eventually delivers you the HTTP response should also emit progress signals that you may observe if you wish.

- *Simple promises* – Do not implicitly "chain" callbacks.

SIMPLE PROMISES

This last problem deserves special attention. The rest are mere annoyances and could be suffered through, if not for implicit chaining. It is a fundamental design flaw, and I wouldn't be surprised to learn that it's responsible for more bugs in Twisted-using programs than any other single factor.

Let me first spell out exactly what I mean here by "implicitly chained" callbacks and "simple" promises. In Twisted, you can write:

```
deferred = background_job()
deferred.addCallback(cb_1)
deferred.addCallback(cb_2)
deferred.addCallback(cb_3)
```

Each callback here will be given the **return value** of the previous callback. I'll refer to that as *implicit chaining*.

Instead I advocate having the promise give each callback simply the **same value** – the original result of the background job. So I'll call this a *simple promise*. (In these examples, I'll use `deferred` for objects with implicit chaining and `promise` for simple promises.)

```
promise = background_job()
promise.addCallback(cb_a)
promise.addCallback(cb_b)
promise.addCallback(cb_c)
```

In this example, each callback will get the exact same value.

Nothing that any one of them does can affect the others.

Simple promises are more general. The key is to have `addCallback` and its friends return a *new* promise for the result of the callback. With this feature, you can still chain callbacks, but you must do it explicitly. That is a good thing. Consider a deferred with implicit chaining:

```
def add4(n):
  return n + 4

deferred = background_job()
deferred.addCallback(add4)
deferred.addCallback(log)
```

Supposing `background_job` supplies a value of 3, this example will log `7`. We can just as easily do that without implicit chaining:

```
def add4(n):
  return n + 4

promise = background_job()
promise2 = promise.addCallback(add4)
promise2.addCallback(log)
```

This also logs `7`. Now let's look at an example starting with a simple promise:

```
promise = background_job()
promise.addCallback(log)
promise.addCallback(log)
```

This logs `3`, twice. But try doing this with implicit chaining. It can't be expressed. (Yes, you could achieve the same output in many different ways, but here I'm concerned with the structure of control flow.)

More importantly, implicitly chained callbacks are confusing. You must pay careful attention to the order in which you add callbacks. They require complicated diagrams to explain how they behave. If

you want to insert a new callback somewhere, you have to be extra careful when you do it, to ensure it goes in the right place in the chain. By contrast, if you want to insert a new callback somewhere with simple promises, you have only to stick it on the correct promise.

Further, implicit chaining makes you do extra work to propagate return values and errors, even when your callback properly doesn't care about such things. Let's say you have the following snippet (which is the same for either a `promise` or a `deferred`):

```
either = background_job()
either.addCallbacks(my_result, my_error_handler)
```

You just want some basic logging to check what's going on. With a simple promise, that's easy:

```
promise = background_job()
promise.addCallback(log)
promise.addCallbacks(my_result, my_error_handler)
```

With implicit chaining, it's more work:

```
def safe_identity_log(x):
  try:
    log(x)

  # If log raises an exception, we still
  # want our real callback to fire, so we
  # have to catch everything here, even
  # though that has nothing to do with the
  # function of this callback.
  except:
    pass

  # Likewise, we must take care to return
  # the original value, or else the
  # callback will just get None.
  return x

deferred = background_job()
deferred.addCallback(safe_identity_log)
deferred.addCallbacks(my_result, my_error_handler)
```

THIS POST IS TOO LONG

Anyway. That's all I got. I really want to see this exist. So badly, I might actually do it myself. But it will have to wait a bit.

ADDENDUM: COROUTINES AND CONTINUATIONS

Writing good code in most asynchronous systems (including Twisted, node.js, and even E) feels inside-out. Your results are passed *in* as parameters; they don't come *out* as return values like they normally would. Same for exceptions. This causes more verbosity, and it just *feels weird*.

My earlier post The Wormhole describes a transformation that turns things right-side out again. (It's built out of continuations, but it could just as well be done with coroutines, say in Python.) It makes writing correct asynchronous code almost as easy as writing correct synchronous code. However, it can only be done correctly if your promises are of the simple variety. I've since learned that Twisted has attempted this trick. That implementation is useful, but it has several sharp corners. For example, this will not do what you would hope:

```
background_deferred = None
can_background_job_complete = False

@deferred.inlineCallbacks
def f():
  global background_deferred
  background_deferred = background_job()
  value = yield background_deferred
  returnValue(value + 4)

final_deferred = f()
background_deferred.addCallback(log)
final_deferred.addCallback(log)
can_background_job_complete = True
```

Supposing `background_job` supplies 3, what will this log? In real life: `None`, then `7`. If these were simple promises, it would log `3`, then `7`.

COMMENTS