Concurrent network programming with Twisted

Release 0.6

Jean Daniel Browne

April 27, 2010

Contents

1 What is Twisted?

Twisted is a an organised set of Python modules, classes and functions aiming at efficiently building client or server applications. Twisted base classes wrap the UDP, TCP and SSL transport protocols and child classes of the tranport classes offer well tested, application protocol implementations which can coherently weave file tranfer, email, chat and presence, enterprise messaging, name services, etc. This *concurrent network framework* enables developers to produce stable and performant mail client or domain name servers with less than fifteen lines of code.

Also, Twisted has methods for implementing features which are often required by sophisticated software projects. For instance, Twisted can map a tree of ressources behind URLs, can authentify users against flexible backends, or can safely distribute objects on a network enabling remote procedure calls.

Twisted is written in Python and C and is fast partly because, as we will see, the data is processed as soon as it is available no matter how many connections are open. Twisted is used in production in the datacenters of the Canonical or Fluendo corporations, for example, but also as the IO engine of the Zope Toolkit or embedded by smaller applications to act as the HTTP layer of the desktop edition of the Moinmoin wiki.

Twisted presents two difficulties: it requires to think the application aound the core concepts detailed below. Incremental project migration to Twisted is not straightforward. Second point, Twisted does not make a seamless use of multicore, the application must be designed to use multiple processes (Twisted offer helpers functions though).

This article introduces the problem of network concurrency over sequential programming, then compares Twisted's model to the classical thread+socket model, to point out that the classical model is less efficient ofr a high number of connections. Follows an overview of the main objects of the framework, namely the *reactor* which supervises the simultaneous network connections, the *protocols* and the *deferreds*. Finally, to get a feel of Twisted code, two web client scripts are compared: one sequential and one concurrent. Then, as we explain new Twisted key concepts, the concurrent script is improved.

2 Presentation of the problem

Network concurrency is a key concept particularly for performance: take a simple problem such as retrieving, for each blog of a list of blogs, the title of the web page of the first article of the blog. This first problem is actually the core job of a scraper or a crawler. This means:

```
for each blog url
    retrieve the list of articles
    get the first article url in the list
    retrieve the web page of the first article
    display the title
```

Let's provide a quick and naive solution to this problem. Here are three handy functions :

- **urlopen**(url) sends an HTTP GET request to the url and returns the body of the HTTP response as an open file,
- **parse**(HTML string) takes an HTML string as an input and returns a tree structure of HTML nodes,
- htmltree.**xpath**(pattern) returns a list of nodes matching the pattern. The text content of a HTML node is accessed via the member attribute `.text`. We will use **xpath** to find urls or page titles in a HTML document.

And here is the script which brings all this together (and includes a design problem):

```
# sequential.py
from lxml.html import parse
from urllib2 import urlopen

for planet in ["http://planet.debian.net",
               "http://planetzope.org",
               "http://planet.gnome.org",
               "http://gstreamer.freedesktop.org/planet/"]:

    # first Xpath pattern matches articles links, second pattern: html titles
    article = parse(urlopen(planet )).xpath('//h3/a/@href'    )[0]
    title   = parse(urlopen(article)).xpath('/html/head/title')[0].text

    print "first article on %s : \n%s\n%s\n\n" % (planet, article, title )
```

When there are *n* element in the blog list, there will be *2n* page downloaded, one after the other, and this will take *2n * time to download a page*. When the time taken by an algorithm the algo directly proportional to the number of inputs, this is called a linear complexity and this will rightfully raise the eyebrow of any developer concerned with performance and scalability.

As each download is completely independent from each other, it is obvious that these downloads should be executed in parallel, or, *concurrently*, and this is the raison d'être of the Twisted Python framework. Processes and threads are well-known primitives for programming concurrently but Twisted does without (not even behind your back), because it is not adapted for scalable network programming. This frees the developer from using semaphores, mutexes or recursive locks. The solution presented at the end of the article does not have more line of codes, does not take much longer for *n* downloads than it takes for one download (a constant complexity) and is actually three times faster.

Note: A frequently heard reaction at this point is "Python is a slow language to start with, **a fast language** is the answer to performance". Notwithstanding the many existing techniques to make Python code compile and run on multiple processors, the speed of the language is not the point. In many case, even a C compiler can not fix a bad design. For example, take the download of an install CD, there is an insignificant gain in performance in a download client written in C over an implementation in Python, because 1. both implementations are very likely to end up leaving the network and disk stuff to the kernel and most importantly because 2. this job is inherently bound by the network bandwidth, not by CPU computations, where C shines. Both in C and in Python, in the context of multiple downloads, prformance depends on concurrent connections.

One of the core ideas is that Twisted functions which make a network call should not block the application while the response is not yet available. Functions are split in two functions, one which emits the network system call and another function, the *callback*, which will process the received bytes, and then return a processed result. In the period of time between the return of the requesting functions and the execution of the callback, other instructions can be processed. This is the basic idea which makes asynchronous code faster than blocking code.

3 Twisted's network concurrency model as compared with sockets and threads

Twisted's concurrency model of is called *cooperative multitasking* and is really different from a traditional "one socket in one thread" scheduler. Twisted has three advantages over the thread model: it is safer, faster and faster. The advantages are illustrated around a second simple problem: it is the traditional example of a global counter updated by many threads.

3.1 Safer: no need to worry about locking shared ressources

A thread scheduler decides the execution of a thread for a time slice, then pauses the thread at an unpredictable point of a computation to let another thread run. This is problematic, see the following code (using threads) which is incorrect.

```
>>> counter = 0
>>> def incr(n=0):
...     global counter
...     for i in range(1000):
...         counter += 1
```

The *incr* function increments the counter one thousand times. Below, the *execute* function creates 100 threads to run the *incr* function.

```
>>> from threading import Thread
>>> def execute(f):
...     threads=[]
...     for i in range(100):
```

```
...         threads.append(Thread(target=f))
...
...     for t in threads: t.start()
...     for t in threads: t.join()


>>> execute(incr)
>>> counter > 0
True
```

When the threads are run, the counter has been incremented but curiously, it is different from *100 threads * 1000 increments = 100000*.

```
>>> counter == 100000
False
```

The value of counter was 96733 last time this article was checked, which means that 3% of the counter increments went wrong. Here is what happened:

1. thread *x* reads the counter: say 5000, then gets paused
2. thread *y* reads the counter: 5000, increments, then writes the data : 5001
3. thread *x* continues on from where it paused : increments and writes the counter: 5001. Thread y incrementation was missed.

From the Python virtual machine, via the libc, down to the processor instructions, an increment is composed of a variable read and an addition and is not atomic by default. To avoid the effect of a big blind chainsaw messing with a subtle variable increment, threads must use defensive techniques: they define *critical sections* using locks and refuse to enter one until every other thread has left the critical section. Here is a correct version of the *incr* version using a lock dedicated to the *counter* ressource.

```
>>> from threading import Lock
>>> lock = Lock()
>>> def safe_incr():
...     global counter,lock
...     for i in range(1000):
...         with lock:
...             counter+=1
>>> counter = 0
>>> execute(safe_incr)
```

At this point, the counter is correct:

```
>>> counter
100000
```

Note that the class:*Lock* classs abide by the *with* interface `with`. with will dutifully call `Lock.__enter__()` when entering the indented block below, and similarly will call `Lock.__exit__()`, when exiting the block. These functions are implementes to respectively *acquire* and *release* the `Lock` instance (*even* if the block is exited via an exception!).

Shared ressources and critical sections must be controlled by code which is difficult to get it right. Twisted do without: it handles many network connections concurrently and it can hold many pending actions concurrently, but functions and method are not executed concurrently: the callbacks triggered on the various events of a network connection lifecycle will execute one after the other, no matter how many connections exist. Since functions are not concurrent and always execute until they return, there is no race conditions and no need for the definition of critical section with mutexes, recursive locks, etc.

Twisted network concurrency model is called *cooperative multitasking* in the sense that all functions are written striving to return as fast as they can, especially after having emitted a network request.

As the thread scheduler can be compared to a blind chainsaw, Twisted functions are more like relay sprinters who choose when to pass the baton. They decide to pass the baton to the coach who, at the time when he gets the baton, decides which sprinters is the fittest to run. If a sprinter keeps the baton indefinitely, there is no one to interrupt him, and the other sprinters do not get to run: Twisted concurrent model is safer as long as everyone behave as a gentlemen.

3.2 Faster: no overhead scheduling the threads

The previous threaded code using a shared ressource is less and less efficient as the number of threads increases: the ressource becomes a bottleneck. The following function times the execution in multiple threads of the function given as a parameter:

```
>>> from timeit import Timer
>>> chrono = lambda f: Timer(lambda :execute(f)).timeit(number=10)
```

Now, let's compare the execution time of the *incr* and *safe_incr*.

```
>>> no_lock = chrono(incr)
>>> locked  = chrono(safe_incr)
>>> 10 * no_lock < locked
True
```

In our example, the safe and locked code is at least 10 times less efficient. The decision by the OS thread scheduler to run a particular thread is based on an algorithm which does not have much of a clue of the existing ressources. What happens here is when a thread is run, the thread context and stack is copied back which costs CPU cycles and data transfer, and it might actually be in vain, as the thread does not have the lock needed to execute. The threads get re-scheduled until it can own the ressource.

The concept of *ressources* applies equally well to lock ownership, to data received via sockets, or even to user input in a graphical user interface.

As each Twisted function runs until its return point without interruption, there is no concurrent access to the ressource, the ressource is always free for a function which runs. The scheduling overhead does not occur with Twisted. I assume the reader is curious to see the equivalent Twisted code and its performance, I promise to show it as soon as the required Twisted concept have been presented (spoiler: Twisted is twice faster than even the fast and unsafe code).

3.3 Faster: data received does not sit in a buffer while a thread is paused

Once the data is received by the kernel and made available to the application via a file descriptor, the data might actually sit there until the thread which takes care of this file descriptor gets a chance to run again. Event driven frameworks can alleviate this problem, the next section introduces the `reactor` and the `Protocol`, which are a pre-requisite to understand how Twisted solve this delay.

4 The Reactor and the Protocols

How does Twisted do away with the problems of threads in the context of network connections? The Twisted runs a main loop called the reactor which schedules the callbacks. It is the *coach* of our prior comparison. **The reactor scheduling decisions derives directly from the availability of the data received in the supervised file descriptors**. The reactor is twofold:

- it is a wrapper around a specialised system calls which monitor events on an array of sockets. Instead of supervising the sockets from userland, Twisted delegates this hard work to the kernel, via the best system call available on the platform: epoll on Linux, kqueue on BSD, etc

  In a nutshell, these system calls returns after either a timeout or after the reception of data in one of the transport. The system call returns an array of events received, for each supervised file descriptor[1].

  There is a big bonus for a developer to be able to leverage the efficient advanced system calls of multiple kernels with one code. Another bonus is the delegation of the concurrent supervision of the sockets to the kernel. If the kernel offers to do it, why should we re-invent the wheel in userland?

- the reactor maintains a list of Twisted `Protocol` instances, each associated to the sockets registered to *epoll*. When *epoll* returns, one after the other, for each data received in the socket, the reactor dutifully runs the `dataReceived()` of the `protocol` associated to the socket.

The reactor is the runtime hub of the Twisted framework, it handles the network connections and triggers the processing of the received data as soon as they arrive by calling specific methods of the `Protocol` associated to the socket. Let's focus on a single page download, first, with the sequential `urlopen()` function:

1. urlopen parses the domain name from the URL and resolves it to an IP address (this blocking network request may be avoided if the local resolver maintains the domain name in a cache).
2. An HTTP get request for the URL is formatted, a socket toward the IP address of the web server is opened and the message is written in the socket's file descriptor. `urlopen()` waits for the reply from the server and returns.

Here is the corresponding steps of how Twisted operates with the `getPage()` function:

1. `getPage()` parses the input URL, format the HTTP request string, and uses the `reactor.connectTCP()` method to stack a socket creation and monitoring request to the reactor. The argument of `connectTCP()` are a host, a port and an instance of the `HTTPGetClient` class, deriving from the `Protocol` class.

   `connectTCP()` tranparently inserts a DNS request if the host is a domain name and not an IP address. This conditions the HTTP request to the availability of the IP address, in a non blocking manner,

2. `getPage()` returns a deferred, a slot that the developer must fill with a function which will be executed when the HTTP reply arrives (more on the deferred in the next *section*). This function should expect the HTML body of the response as the argument,

3. the reactor is run: for each `Protocol` object queued: the reactor opens a socket, copies the corresponding file descriptor in the `transport` attribute of the `Protocol` instance, and puts the socket under supervision.

   The reactor calls the `connectionMade()` method of the `Protocol` instance which, in the case of `getPage()` writes the formatted HTTP request to the `transport` and returns immediately to the reactor loop,

4. when the reactor detects the reply bytes in the socket associated to `transport`, it calls the `dataReceived()` method of the associated `Protocol` which, in the case of `getPage()`, is written to parse the HTTP header from the HTML body.

   Finally, the `dataReceived()` method for this protocol *fires* the developer callback attached to the instance deferred, with the HTML as the parameter.

Additional abstractions such as the `Factory` interface are left out in this article to ease the learning curve , they are are described in the official documentation. For our third problem, let's compare two complete versions, one concurrent, one sequential of a simple script which, 30 times, prints the HTML title of the *http://twistedmatrix.com* web site.

[1] the C10K problem is a reference on server handling concurrently ten thousands of clients.

```
# trivial_sequential.py
from lxml.html import parse
from urllib2 import urlopen

url = 'http://twistedmatrix.com'

def title(url):
    print parse(urlopen(url)).xpath('/html/head/title')[0].text

# let's download the page 30 times
for i in range(30):
    title(url)
```

Note that in the following version, the Twisted main loop started by `reactor.run()` never returns: a line of code below the start of the reactor loop will never be executed.

```
# trivial_deferred.py
from twisted.internet import reactor
from twisted.internet.defer import DeferredList
from twisted.web.client import getPage
from lxml.html import fromstring

url= 'http://twistedmatrix.com'

def getpage_callback(html):
    print fromstring(html).xpath( '/html/head/title' )[0].text

# 30 pending asynchronous network calls, and attachment of the callback
[ getPage(url).addCallback(getpage_callback) for i in range(30) ]


reactor.run()      # open the network connections, and fires the callbacks
                   # as soon as the replies are available

# Use Ctrl-C to terminate the script
```

The attention should be drawn on the following blocking snippet:

```
html = urlopen(url))
print  parse(html).xpath( ... )
```

which becomes, with Twisted primitives:

```
def getpage_callback(html):
    parse(html).xpath( ... )

getPage(url).addCallback(getpage_callback)
```

It is indeed bewildering to realize that in Twisted, **the calling function can not manipulate the result of the request**. Here is a longer form, which might seem simpler to read because the callback code is presented after the request code:

```
d = getPage(url)
def getpage_callback(html):
    parse(html).xpath( ... )

d.addCallback(getpage_callback)
```

If you don't like neither these style, stay tuned, you will appreciate the section *yield simplifies Twisted code*. There is something unexplained in the last code snippet: what is the object to which *d* is bound? What does `getPage()` returns if it's not the server reply? you will find out in the next section.

5 The *Deferred*

5.1 Advantages and usage

Event driven frameworks are usually provided with a set of classes with predefined events. For example, to model an HTTP client, we expect to have to derive a class and implement a method with a specific name. Something like:

```
class MyClient(HTTPClient):
    gotHtml(html):
        "here my specific client code parsing the html"
```

Twisted indeed provides similar pattern, but Twisted also introduces a powerful abstraction to represent an event and its pending action: the `Deferred` is an object which can holds a function. The code creating a request is expected to return a result, which is unavailable at this point, so instead, it returns a deferred, for which the requesting code expect the user to be filled it with a function to process the results. The requester object which is usually an instance of child class of `Protocol` also keeps a reference to this deferred and should call the callback, as soon as it is notified by the reactor that the data is received. The Twisted documentation calls it a "promise of a result", here and there. Here are three great things about the Deferred:

- avoid the requirement to subclass anything to write a callback. No need for the object oriented programming to kick in, good old functions will do just fine.
- the code making a request does not have to specify, know or care about the name of the callback function, which simplifies the writing of new requesting API. The requester calls the method `callback()` on a deferred, when the data is received. It is up to the user to store the callable it seems adapted, in the Deferred return by asynchronous function.

  It is up to the job of the protocol implementer to create a deferred, keep it as a attribute of the protocol instance and execute the callback which has been set by the protocol user, on this deferred on the desired event.
- the event represented by the deferred, and the pending action it fires can be manipulated: stored, listed, passed around, chained or cancelled. Take a list of events, it is not difficult to set a callback when the first event, or all events have happened.

5.2 Synchronisation

Synchronizing calls means specifying the order and the event at which actions will take place. In a sequential script, the execution schema is implicit and so obvious that it is not even worth mentioning it:

- the network calls are executed along with the successive `urlopen()` function calls
- and the program stops when the interpreter reaches the end of the script.

So far so good, but now, in a Twisted program, things go differently, there is no more gravity, and there is a fifth dimension... ok, I am being a bit dramatic, the differences are more subtle. There are two phases:

1. the first phase is the specification of the execution steps through the stacking of connections request to the reactor, and the definition of callbacks path. `getPage()` function call does not actually trigger a network HTTP request but creates a deferred which stacks a step in a callback chain,
2. the second phase is inside `reactor.run()` , which triggers the execution of the callback chains and synchronizes the callbacks depending on when the response are available.

Just comment out the call to run the reactor in the concurrent script, and use wireshark to check that `getPage()` does not carry out the network call by itself.

In our last problem, the concurrent script did not stop when the 30 calls completed successfully and require an explicit signal to terminate. Let's synchronize the end of the script to the completion of the 30 page download. In Twisted terms, this translates as *gather the deferred returned from the requests in a list, define a callback which will stop the reactor when all the deferreds in the list have completed*.

The code should be modified to create a *DeferredList* from the list of calls to the title function. *DeferredList* is a Twisted primitive which returns a deferred which *fires* when all the deferred have completed. An anonymous function which stop the reactor is attached as a callback to the *DeferredList*:

```
l = [ getPage(url).addCallback(getpage_callback) for i in range(30) ]
d.DeferredList(l)
d.addCallback(lambda n:reactor.stop())
```

The `lambda` return a function whose only action is to call the `reactor.stop()`. This new function is required because `reactor.stop()` does not comply with the callback specification: *a callback must have at least one argument*. The anonymous function created with `lambda` presents the correct signature.

Now that the script terminates gracefully, let's clarify a common misunderstanding: what does the reactor know about the deferreds that the user manipulate? The answer is: nothing. The interfaces that the reactor knows are the few hardcoded functions from the UDP, TCP and SSL transport protocols such as `connectionMade()`, `dataReceived()`, and other methods. The reactor maintains a list of transport instances stored as attributes of protocoles instances which hold a Deferred created by the request methods and that the `dataReceived()` methods expects to fire the callback.

Now this concurrent version terminates, its performance can be compared to a sequential script. It is much more efficient (on my machine, it is 8 times more efficient). Note that for a threaded version of the script:

```
~$ time python trivial_sequential.py
real 1m22.945s
~$ time python trivial_concurrent.py
real 0m10.315s
```

Deferreds do not require a reactor, to show it here is the equivalent of the threaded *counter* code presented above:

```
>>> from datetime import datetime as n
>>> from twisted.internet.defer import Deferred
>>>
>>> counter, start = 0, n.now()
>>> deferreds = [Deferred().addCallback(incr) for i in xrange(100)]
>>> # There is a hundred concurrent pending actions at this point ...
>>>
>>> # ... NOW !
>>> for d in deferreds:
...     d.callback(None)
...
>>> elapsed = n.now() - start
>>> 2 * elapsed.seconds < no_lock
True
>>> counter
100000
```

This code runs even twice faster as the code running 100 threads without locks, and is has the noticeable advantages of being correct.

The central mechanisms of Twisted were presented in the previous sections, you are almost there ! The last section before the conclusion shows a nicer way to present Twisted code. The two first subsections are recap on the standard `yield` keyword and Python decorators.

6 *yield* simplifies Twisted code

... once you understand what this crazy statement does

6.1 the *yield* Python keyword

Python offers a really powerful keyword which Twisted uses in a clever way to simplify the boilerplate of deferred and callback manipulation. `yield` allows for returning from a function half-way through and restarting later on at the point where the function returned. The arguments of `yield` are returned to the caller of the function as if the `return` statement was used. If you already know `yield`, just skip to the next section.

These examples only include code from the core Python language, there is no Twistery involved:

```
>>> def func_with_several_entry_points():
...     yield 'this string is the first return value'
...     val = 1+1
...     yield 'the latest portion of the function was executed',val
...
>>> f=func_with_several_entry_points()
>>> f                                    # doctest:+ELLIPSIS
<generator object func_with_several_entry_points at ...>
>>> f.next()
'this string is the first return value'
>>> f.next()
('the latest portion of the function was executed', 2)
>>> f.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

On call, a function using `yield` returns a Python *generator* object i.e. an object with a `next()` method which, on successive calls, runs the sections of code delimited by the `yield` statement, one after the other. A generator object also raises a `StopIteration` exception to signal when it has reached the end of the last code section, and that it is no use calling it again.

`Yield` is really powerful: for instance, here is a *lazy* implementation of the fibonacci suite.

```
>>> def fib(max=10):
...     a,b=1,0
...     for i in range(max):
...         yield b
...         b,a = a+b,b
```

Lazy in the sense that it behaves like a huge list but the whole list is never completely computed in one shot and never fully stored in memory: the next element is computed **on demand**, when the `next()` method is called:

```
>>> gen=fib(2)
>>> gen.next()
0
>>> gen.next()
```

```
1
>>> gen.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Generators are integrated with the `for` keyword which dutifully call the `next()` method on and on, until the `for` keyword catches the `StopIteration` exception:

```
>>> [n for n in fib(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
```

But we digress: now back to Twisted, do you see the similarity of concept between the functions using `yield` and the Twisted chains of callback ? *Both specify section of codes to be called successively*.

A limitation of `yield` mechanism was lifted in Python2.5, which makes it usable from the reactor: the next section of code can be called with input data with the new `send()` method instead of `next()`. `yield` must be used on the right hand side of a variable binding (the *equal* sign), the sent data is bound to the variable.

```
>>> def func():
...     data = yield "Ok, I am ready to receive data"
...     yield "The double of the data I just received", 2*data
...
>>> t=func()
>>> t.next()
'Ok, I am ready to receive data'
>>> t.send('Hello')
('The double of the data I just received', 'HelloHello')
```

These changes turn generators from one-way producers of information into both producers and consumers. The reactor can build generators which send a network request the first time they are called, and can *send* the generator the response data for processing, when it is available.

6.2 Decorators in Python

Twisted uses the *decorator syntax* to write callbacks in simpler manner, this section is just a brief recap of what is a decorator, skip to the next section if comfortable with Python decorators.

A decorator is a function returning another function, usually applied as a function transformation. For example, it is useful when you want to debug a series of nested calls, such as

```
parse(urlopen(url))
```

If there is a need to know what was returned by urlopen *without modifying the nested call*, a solution is to insert the following statement at the previous line:

```
parse = log(parse)
parse(urlopen(url))
```

Where `log()` is defined as:

```
>>> def log(f):
...     def foo(n):
...         print "Here is the argument:", n
```

```
...         return f(n)
...     return foo
```

`log` prints the argument, then `log` calls the decorated function and return the result. In our example, the HTML string will be printed before being passed on to the parse function. Here on a custom function:

```
>>> def double(n):
...     return 2*n
...
>>> double=log(double)
```

Python allows some syntactic sugar, with the use of the @ character, for applying a decorator on a custom function to simplify the function definition above:

```
>>> @log
... def double(n):
...     return 2*n
...
```

Both definitions are equivalent:

```
>>> double(5)
Here is the argument: 5
10
```

Now that the yield statement and the decoration syntax are clearer, understanding the integration of yield with the Twisted reactor should be straightforward.

6.3 The integration of *yield* with the Twisted main loop

The Twisted technical constraint to manipulate the result of a request in a function different than the function making the request can be inconvenient: the integration of `yield` with the `reactor` alleviates this problem. Here are two versions of the `title()` scraping function:

```
def title(url):
    d = getPage(url)
    def getpage_callback(html):
        print parse(html).xpath( ... )
    d.addCallback(getpage_callback)
```

The second one is a rewrite with the `yield` statement:

```
@inlineCallbacks
def title(url):
    html = yield getPage(url)
    print fromstring(html).xpath( '/...' )
```

Because `title()` is marked with the `inlineCallbacks()` decorator, it will store a generator and return a deferred, the `reactor` will trigger the call to the `send()` method on the generator, with the requested HTML page as the argument.

This version is shorter, there is no need to create and name a nested function, and to add a level of indentation to the callback code. The code appear more like its sequential counterpart.

7 A concurrent solution to our original problem

Here is a concurrent solution to the problem detailed in the introduction. It is three times faster than the sequential approach :

```
# concurrent.py
from twisted.internet import reactor
from twisted.internet.defer import DeferredList, inlineCallbacks
from twisted.web.client import getPage
from lxml.html import fromstring

@inlineCallbacks
def first_title(url):

    html = yield getPage(url)
    article = fromstring(html).xpath('//h3/a/@href')[0]

    html = yield getPage(article)
    title = fromstring(html).xpath('/html/head/title')[0].text

    print "first article on %s : \n%s\n%s\n\n" % (url, article, title)

planets = ["http://planet.debian.net",
           "http://planetzope.org",
           "http://planet.gnome.org",
           "http://gstreamer.freedesktop.org/planet/"]

d = DeferredList([first_title( p ) for p in planets])
d.addCallback(lambda _:reactor.stop())

reactor.run()
```

The Twisted equivalent of `urlopen()` is called `getPage()`. It is asynchronous and returns a deferred. The low level steps composing `getPage()` are asynchronous as well: even the DNS request turning the url argument into an IP address will not block the application which is why such code is efficient.

This article leaves many questions aside. For instance, error handling is non existent in the scripts: manipulating deferreds explicitly, though more verbose, help creating clearer failure code path and help create more robust application and libraries. In our script, as well as when building network applications or libraries, the following problems may arise: no network, no dns, no route, no tcp server, page not found error, HTML title not found. How easy it is to handle them gracefully?