LECTURE 12

Twisted

TWISTED

Last lecture, we introduced the mental model needed to work with Twisted. The most important points to keep in mind are:

- Twisted is asynchronous. Multiple independent tasks may be interleaved within a single thread of control in order to avoid wasting time during blocking operations.
  - One task running at a time → less complexity.
  - Yielding control on blocking operations → time efficient.
- Twisted is event-driven. The program flow is determined by external events and managed by a reactor loop.

We're going to start with toy examples to learn the elements of twisted. Then, we'll write a simple TCP echo server with Twisted.

INSTALLING TWISTED

Twisted is the first library we’ll be working with that does not come with the Python Standard Library.

However, it might already be installed on Ubuntu! Try the following in the interpreter:

```
>>> import twisted
>>> twisted.__version__
‘13.2.0’
```

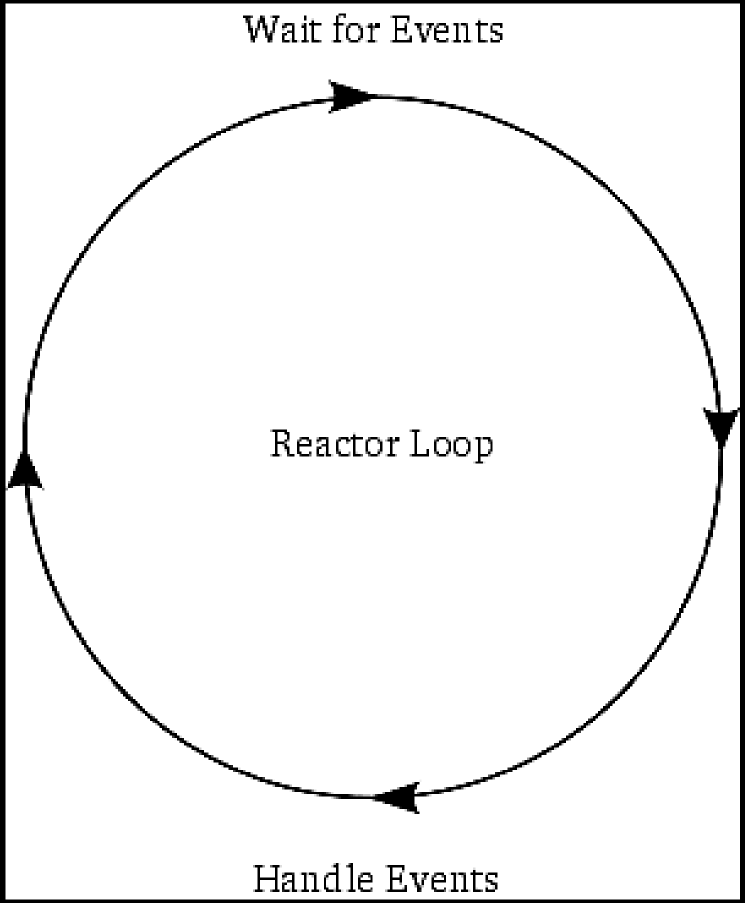
If that doesn’t work, then issue the following command:

```
$ sudo apt-get install python-twisted
```

TWISTED REACTOR

As we mentioned last lecture, at the heart of every twisted application is a single reactor loop.

The reactor is responsible for listening for and responding to network, filesystem and timer events. The term "responding" in this context means dispatching events to event handlers.



TWISTED REACTOR

Creating a reactor couldn't be simpler.

```
    from twisted.internet import reactor
```

This import statement not only makes the reactor name available to us but also initializes the singleton reactor that will be used throughout our entire Twisted application.

TWISTED REACTOR

A reactor has a number of core methods available.

```
from twisted.internet import reactor

reactor.run()  # Start the reactor
reactor.stop()  # Stop the reactor
```

The simplest are `run()` and `stop()` which simply start and stop the reactor. Note that in this simple code, the reactor has no scheduled event and no system triggers. This reactor will block on doing nothing. In fact, it won't even stop. Why would that be?

TWISTED REACTOR

To actually get a reactor that does something, we could use one of the following methods:

- `callWhenRunning(callable, *args, **kwargs)` – Call a function when the reactor is running. If the reactor has not started, the callable will be scheduled to run when it does start. Otherwise, the callable will be invoked immediately.
- `callLater(delay, callable, *args, **kwargs)` – Call a function *delay* seconds later.

TWISTED REACTOR

```python
import sys
from twisted.python import log
log.startLogging(sys.stdout)

def func(x):
    log.msg("In func as event handler")
    log.msg(str(x))
    log.msg("Shutting down now!")
    reactor.stop()

from twisted.internet import reactor
reactor.callWhenRunning(func, "Hello!")
reactor.run()
```

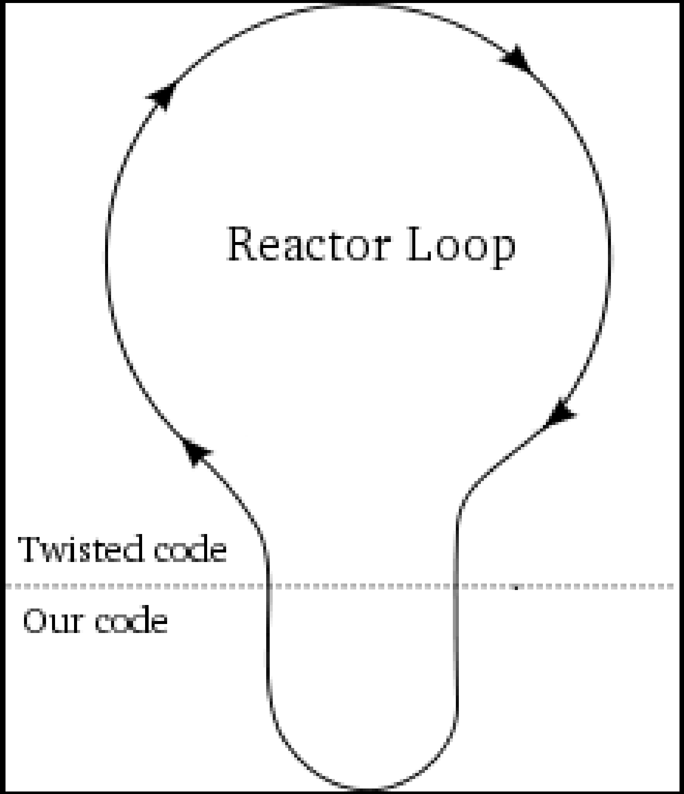
TWISTED REACTOR

The reactor schedules a call to our function func for when it starts running. At this point, it yields control to our code until we explicitly return control back.

```python
import sys
from twisted.python import log
log.startLogging(sys.stdout)

def func(x):
    log.msg("In func as event handler")
    log.msg(str(x))
    log.msg("Shutting down now!")
    reactor.stop()

from twisted.internet import reactor
reactor.callWhenRunning(func, "Hello!")
reactor.run()
```



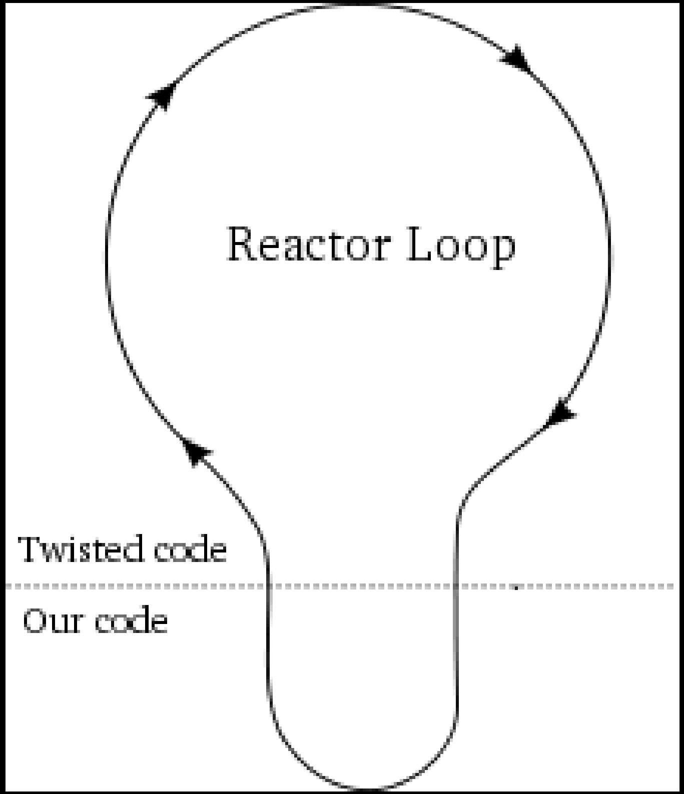
TWISTED REACTOR

We have no blocking operations in our code so we perform our task, and shutdown the reactor.

```python
import sys
from twisted.python import log
log.startLogging(sys.stdout)

def func(x):
    log.msg("In func as event handler")
    log.msg(str(x))
    log.msg("Shutting down now!")
    reactor.stop()

from twisted.internet import reactor
reactor.callWhenRunning(func, "Hello!")
reactor.run()
```



TWISTED REACTOR

```
import sys
from twisted.python import log
log.startLogging(sys.stdout)

def func(x):
    log.msg("In func as event handler")
    log.msg(str(x))
    log.msg("Shutting down now!")
    reactor.stop()

from twisted.internet import reactor
reactor.callWhenRunning(func, "Hello!")
reactor.run()
```

```
$ python twist.py
2015-02-17 11:46:16-0500 [-] Log opened.
2015-02-17 11:46:16-0500 [-] In func as event handler
2015-02-17 11:46:16-0500 [-] Hello!
2015-02-17 11:46:16-0500 [-] Shutting down now!
2015-02-17 11:46:16-0500 [-] Main loop terminated
```

TWISTED REACTOR

Another example using the callLater method.

```
import sys, time
from twisted.python import log
log.startLogging(sys.stdout)

def func(x):
    log.msg(str(x))
    now = time.localtime(time.time())
    log.msg(str(time.strftime("%y/%m/%d %H:%M:%S", now)))
    log.msg("Shutting down now!")
    reactor.stop()

now = time.localtime(time.time())
log.msg(str(time.strftime("%y/%m/%d %H:%M:%S", now)))
from twisted.internet import reactor
reactor.callLater(5, func, "Hello after 5 seconds!")
reactor.run()
```

TWISTED REACTOR

```
$ python twist.py
2015-02-17 11:49:10-0500 [-] Log opened.
2015-02-17 11:49:10-0500 [-] 15/02/17 11:49:10
2015-02-17 11:49:15-0500 [-] Hello after 5 seconds!
2015-02-17 11:49:15-0500 [-] 15/02/17 11:49:15
2015-02-17 11:49:15-0500 [-] Shutting down now!
2015-02-17 11:49:15-0500 [-] Main loop terminated.
```

```python
import sys, time
from twisted.python import log
log.startLogging(sys.stdout)

def func(x):
    log.msg(str(x))
    now = time.localtime(time.time())
    log.msg(str(time.strftime("%y/%m/%d %H:%M:%S", now)))
    log.msg("Shutting down now!")
    reactor.stop()

now = time.localtime(time.time())
log.msg(str(time.strftime("%y/%m/%d %H:%M:%S", now)))
from twisted.internet import reactor
reactor.callLater(5, func, "Hello after 5 seconds!")
reactor.run()
```

TWISTED REACTOR

So far, we haven’t seen anything that helpful yet. Right now, we just have a reactor loop that schedules function calls. What if I do something like this?

```python
import sys, time
from __future__ import print_function

def func(x):
    print(x)
    time.sleep(30)
    print("30 seconds have passed!")

from twisted.internet import reactor
reactor.callWhenRunning(func, "Hello!")
reactor.callLater(10, print, "10 seconds have passed!")
reactor.run()
```

TWISTED REACTOR

So far, we haven’t seen anything that helpful yet. Right now, we just have a reactor loop that schedules function calls. What if I do something like this?

```
import sys, time
from __future__ import print_function

def func(x):
    print(x)
    time.sleep(30)
    print("30 seconds have passed!")

from twisted.internet import reactor
reactor.callWhenRunning(func, "Hello!")
reactor.callLater(10, print, "10 seconds have passed!")
reactor.run()
```

The reactor will attempt to schedule a call to func at t = 0s and a call to print at t = 10s.

However, func sleeps for 30s – effectively blocking the reactor for no reason at all. We should have func give back control to the reactor so that it can continue doing work.

TWISTED REACTOR

So far, we haven't seen anything that helpful yet. Right now, we just have a reactor loop that schedules function calls. What if I do something like this?

```
import sys, time
from __future__ import print_function

def func(x):
    print(x)
    time.sleep(30)
    print("30 seconds have passed!")

from twisted.internet import reactor
reactor.callWhenRunning(func, "Hello!")
reactor.callLater(10, print, "10 seconds have passed!")
reactor.run()
```

```
$ python twisty.py
Hello!
30 seconds have passed!
10 seconds have passed!
```

TWISTED DEFERREDS

The Twisted docs have an excellent introduction to deferreds and async programming in general (but it's, like, impossible to find). So, I'll reproduce it here.

Assume we have the following lines of Python code:

```
pod_bay_doors.open()
pod.launch()
```

This piece of code relies on a certain order of execution. Of course, the pod bay doors must be open before we can launch the pod. Luckily, once we hit line two, we know that line one has completed.

TWISTED DEFERREDS

The Twisted docs have an excellent introduction to deferreds and async programming in general (but it's, like, impossible to find). So, I'll reproduce it here.

Assume we have the following lines of Python code:

```
pod_bay_doors.open()
pod.launch()
```

In asynchronous programming, however, operations should not block the reactor. If the `pod_bay_doors.open()` method takes a long time, we should immediately return to the reactor with an IOU. This lets the reactor work on other things in the meantime. However, we wouldn't want it to work on `pod.launch()` just yet. What to do?

TWISTED DEFERREDS

Something like this makes more sense for asynchronous programming:

```
placeholder = pod_bay_doors.open()
placeholder.when_done(pod.launch)
```

We're telling the reactor what to do when the results of the call to `pod_bay_doors.open()` are received. We're instructing the reactor on the order of execution in the event that `pod_bay_doors.open()` actually makes good on its IOU.

TWISTED DEFERREDS

In Twisted, this placeholder is a Deferred object. So, instead of the synchronous scheme:

```
pod_bay_doors.open()
pod.launch()
```

We will use the “placeholder” method:

```
d = pod_bay_doors.open()
d.addCallback(lambda ignored: pod.launch())
```

TWISTED DEFERREDS

```
d = pod_bay_doors.open()
d.addCallback(lambda ignored: pod.launch())
```

- `pod_bay_doors.open()` returns a Deferred object, which we assign to `d`.
- The deferred `d` is like a placeholder, representing the return value of `open()`.
- We specify the next thing to do when `open()` eventually finished by adding a callback to `d`.
- Whenever `open()` eventually finishes, the callback will be called.
- In this case, we don’t care what the return value is – we just want to know when it’s done – so we use a meaningless parameter in the callback function.

TWISTED DEFERREDS

Let’s try a little exercise:

```
    sorted(x.get_names())
```

Let’s say that `x.get_names()` is a blocking operation. How can we rewrite this code with Deferreds to makes it asynchronous?

TWISTED DEFERREDS

Let’s try a little exercise:

```
sorted(x.get_names())
```

Let’s say that `x.get_names()` is a blocking operation. How can we rewrite this code with Deferreds to makes it asynchronous?

Note that the above is equivalent to:

```
names = x.get_names()
sorted(names)
```

TWISTED DEFERREDS

```
names = x.get_names()
sorted(names)
```

Since `x.get_names()` is a blocking operation, we should have it return a deferred object (i.e. placeholder or IOU).

The next step is to sort the return value (`names`) so we add the `sorted()` function as a callback.

```
d = x.get_names()
d.addCallback(sorted)
```

The return value `names` is automatically passed as an argument to the `sorted()` function when `x.get_names()` eventually finishes.

TWISTED DEFERREDS

Let's say we wanted to do the following where `get_names()` and `sorted()` are both blocking operations:

```
print(sorted(x.get_names()))
```

This is equivalent to:

```
names = x.get_names()
sorted_names = sorted(names)
print(sorted_names)
```

Which can be written asynchronously as:

```
d = x.get_names()
d.addCallback(sorted)
d.addCallback(print)
```

TWISTED DEFERREDS

Let's say we wanted to do the following where `get_names()` and `sorted()` are both blocking operations:

```
print(sorted(x.get_names()))
```

This is equivalent to:

```
names = x.get_names()
sorted_names = sorted(names)
print(sorted_names)
```

Which can be written asynchronously as:

```
d = x.get_names()
d.addCallback(sorted)
d.addCallback(print)
```

We've just created a callback chain. These statements are equivalent to:

```
x.get_names().addCallback(sorted).addCallback(print)
```

TWISTED DEFERREDS

So, now let’s try to make our blocking example asynchronous. Note that this is sort-of contrived because we don’t actually have a mechanism to fire signals for us so we have to do it ourselves.

```python
import sys, time
from __future__ import print_function

def func(x):
    print(x)
    time.sleep(30)
    print("30 seconds have passed!")

from twisted.internet import reactor
reactor.callWhenRunning(func, "Hello!")
reactor.callLater(10, print, "10 seconds have passed!")
reactor.run()
```

```
$ python twisty.py
Hello!
30 seconds have passed!
10 seconds have passed!
```

TWISTED DEFERREDS

```
from __future__ import print_function
from twisted.internet import defer, reactor
import sys, time

def func(x):
    print(x)
    d = defer.Deferred()
    reactor.callLater(30, d.callback, "30 seconds have passed!")
    return d

d = func("Hello!")
d.addCallback(print)
reactor.callLater(10, print, "10 seconds have passed!")
reactor.run()
```

TWISTED DEFERREDS

```python
from __future__ import print_function
from twisted.internet import defer, reactor
import sys, time

def func(x):
    print(x)
    d = defer.Deferred()
    reactor.callLater(30, d.callback, "30 seconds have passed!")
    return d

d = func("Hello!")
d.addCallback(print)
reactor.callLater(10, print, "10 seconds have passed!")
reactor.run()
```

```
$ python twisty.py
Hello!
10 seconds have passed!
30 seconds have passed!
```

TWISTED DEFERREDS

Let’s get some other abstractions under our belt before we continue with deferreds. They’re a tricky topic so don’t be discouraged if it’s not clear right away. It will be clearer when we look at a fuller example.

TRANSPORT BASICS

A Twisted *Transport* represents a single connection that can send/receive data. Think of it like Twisted's version of a socket object. But much nicer.

The Transport abstraction represents *any* such connection and handles the details of asynchronous I/O for whatever sort of connection it represents. The methods defined are:

- `write(data)` – send some data.
- `writeSequence(list_of_data)` – send a sequence of data.
- `loseConnection()` – close connection.
- `getPeer()` – get remote address of other side of connection.
- `getHost()` – get address of this side of connection.

TRANSPORT BASICS

You may have noticed that there are no methods for *reading* data. The Transport object will make a callback when it receives data – we don't have to explicitly make it happen.

Also, note that these methods are really just suggestions. Remember, Twisted is in control, not us. So when we tell the Transport to write some data, we're really asking it to write some data whenever it is able to.

PROTOCOLS

Protocols implement protocols. This could be one of the built-in protocols in twisted.protocols.basic or one of your own design.

Strictly speaking, a Protocol instance implements the protocol for a single connection. This connection is represented by a Transport object.

So every connection requires its own Protocol instance (and therefore, Protocol is a good candidate for storing stateful information about the connection).

PROTOCOLS

Methods of the Protocol include:

- `dataReceived(data)` – called whenever data is received (transport's callback!)
- `connectionLost(reason)` – called when connection is shut down.
- `makeConnection(transport)` – associates transport with protocol instance to make connection.
- `connectionMade()` – called when a connection is made.

PROTOCOL FACTORIES

Protocol Factories simply create Protocol instances for each individual connection. Of interest is just one method:

- `buildProtocol(`*`addr`*`)`: The `buildProtocol` method is supposed to return a new Protocol instance for the connection to *addr*. Twisted will call this method to establish Protocols for connections.

SIMPLE TCP SERVER

```python
from twisted.internet.protocol import Protocol, Factory

class Echo(Protocol):
    # reactor will call makeConnection with current transport instance.
    def dataReceived(self, data):
        # when transport executes callback
        # with data, just echo
        self.transport.write(data)

f = Factory()
f.protocol = Echo # f.buildProtocol() will create an instance of f.protocol
from twisted.internet import reactor
reactor.listenTCP(9000, f) # listenTCP is a built-in method of reactor
reactor.run()
```

SIMPLE TCP SERVER

Try "telnet localhost 9000".
Anything you send will be echoed back to you.

The only thing we're doing explicitly is running the server. Everything that is executed from that point forward is handled and scheduled by Twisted.

```
from twisted.internet.protocol import Protocol, Factory

class Echo(Protocol):
    # reactor will call makeConnection with current transport instance.
    def dataReceived(self, data):
        # when transport executes callback
        # with data, just echo
        self.transport.write(data)

f = Factory()
f.protocol = Echo # f.buildProtocol() will create an instance of f.protocol
from twisted.internet import reactor
reactor.listenTCP(9000, f) # listenTCP is a built-in method of reactor
reactor.run()
```