LECTURE 11

Networking Part 2

NETWORKING IN PYTHON

At this point, we know how to write a simple TCP client and simple TCP server using Python's raw sockets module. This is enough to manually create a simple networked application. Also, it is valuable to be able to work with sockets because it gives you insight into how a networked application works.

However, more sophisticated networked applications are rarely created directly on top of sockets. Typically, it is more useful to use a higher-level networking library which hides some of those details.

NETWORKING IN PYTHON

Today, we'll look at the SocketServer library which is made to simplify the task of creating servers.

Then, we'll start looking at Twisted. Twisted is a popular event-driven networking engine which supports a variety of protocols. Admittedly, Twisted has a steep learning curve but once you know it, you'll be able to write complex networked applications like a pro.

SOCKETS

First, let's consider a simple TCP server made with raw sockets. What's happening here?

```
from socket import *

s = socket(AF_INET, SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    data = c.recv(1024).strip()
    print "{} wrote: ".format(a[0]),
    print data
    c.send(data.upper())
    c.close()
```

SOCKETS

This simple TCP server simply echoes some data received by a client, but all in uppercase.

```
from socket import *

s = socket(AF_INET, SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    data = c.recv(1024).strip()
    print "{} wrote: ".format(a[0]),
    print data
    c.send(data.upper())
    c.close()
```

SOCKETSERVER

The SocketServer module can be used by including the `import SocketServer` statement at the top of your module.

The SocketServer module defines four basic server classes:

- `SocketServer.TCPServer()` – creates a TCP/IP server object.
- `SocketServer.UDPServer()` – creates a UDP/IP server object.
- `SocketServer.UnixStreamServer()` – creates a TCP server using Unix domain sockets.
- `SocketServer.UnixDatagramServer()` – creates a UDP server using Unix domain sockets.

As with the sockets lesson, we will only concern ourselves with TCP/IP as it is the most common.

SOCKETSERVER

The SocketServer module also defines the `SocketServer.BaseRequestHandler()` class.

The first step to creating a TCP/IP server is to create a request handler class which inherits from `BaseRequestHandler`. We'll override the `BaseRequentHandler.handle()` method, which is used to process incoming requests.

We also have `BaseRequentHandler.request`, the socket connected to the client, available to us as well as `BaseRequestHandler.client_address`, the address of the client.

SOCKETSERVER

Here's a simple little handler which takes in data from a client, and echoes the data back in uppercase format.

```
import SocketServer

class MyTCPHandler(SocketServer.BaseRequestHandler):
    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print "{} wrote: ".format(self.client_address[0]),
        print self.data
        # just send back the same data, but upper-cased
        self.request.send(self.data.upper())
```

SOCKETSERVER

The next step is to create an instance of one of the server classes, passing it the server's address as well as the request handler class.

```
if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    # Create the server, binding to localhost on port 9999
    server = SocketServer.TCPServer((HOST, PORT), MyTCPHandler)
```

SOCKETSERVER

The last step is to either call `handle_request()` to process one request or `serve_forever()` to handle requests as they come in.

```
if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    # Create the server, binding to localhost on port 9999
    server = SocketServer.TCPServer((HOST, PORT), MyTCPHandler)
    # Activate the server; this will keep running until you
    # interrupt the program with Ctrl-C
    server.serve_forever()
```

Now we have a SocketServer equivalent to the raw socket server shown earlier in the lecture. This includes the ability to only handle one connection at a time.

SOCKETSERVER

My new server object has the following attributes available:

- `server.server_address` – the address on which the server is listening.
- `server.request_queue_size` – the number of incoming connections to queue. Default is 5.
- and many others. But the point is that we don't need to worry about these details – SocketServer does it for us.

SOCKETSERVER

Of course, we want to be able to handle more than one connection at a time. Luckily, the SocketServer module makes this very easy.

SocketServer also defines a mix-in class called `ThreadingMixIn`. Mix-in classes are essentially classes that only exist to be inherited with other classes. They are not meant to be used on their own.

```
import SocketServer
class ThreadedTCPServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer):
    pass
```

Now we have a custom TCP server that supports threading. Note that the `ThreadingMixIn` class should always be inherited first as it overrides a method in the `TCPServer` class.

SOCKETSERVER

```
import SocketServer

class ThreadedTCPServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer):
    pass

if __name__ == "__main__":
    HOST, PORT = "localhost", 9000
    server = ThreadedTCPServer((HOST, PORT), MyTCPHandler)
    server.serve_forever()
    # Alternatively, start a thread with the server which
    # will then start one for each request
    # server_thread = threading.Thread(target=server.serve_forever)
    # server_thread.start()
```

BLACKJACK SOCKETSERVER

Check out blackjack_ss_server.py to see a SocketServer version of our blackjack server.

NETWORKING

Python also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, and so on.

Protocol	Function	Port	Python module
HTTP	Web pages	80	httplib, urllib2
FTP	File transfers	20	ftplib
SMTP	Sending email	25	smtplib
POP3	Fetching email	110	poplib
IMAP4	Fetching email	143	imaplib
Telnet	Command lines	23	telnetlib

TWISTED

Twisted is an event-driven networking engine written in Python. It is one of the most popular networking engines. In Twisted's own words, the advantages of creating a Twisted-based server:

- Twisted (and other Python-based servers) are immune to buffer-overflow attacks.
- Extremely stable.
- Supports a wide number of protocols and applications.
- Active community, large base of users.

ASYNCHRONOUS MODEL

At its core, Twisted is meant to implement non-blocking asynchronous servers.

Tasks are scheduled in the framework's execution thread, returning control to its caller immediately and before the completion of the task. An event-driven mechanism communicates the results of the execution.

Before we begin building a Twisted server, let’s learn more about the asynchronous event-driven model.

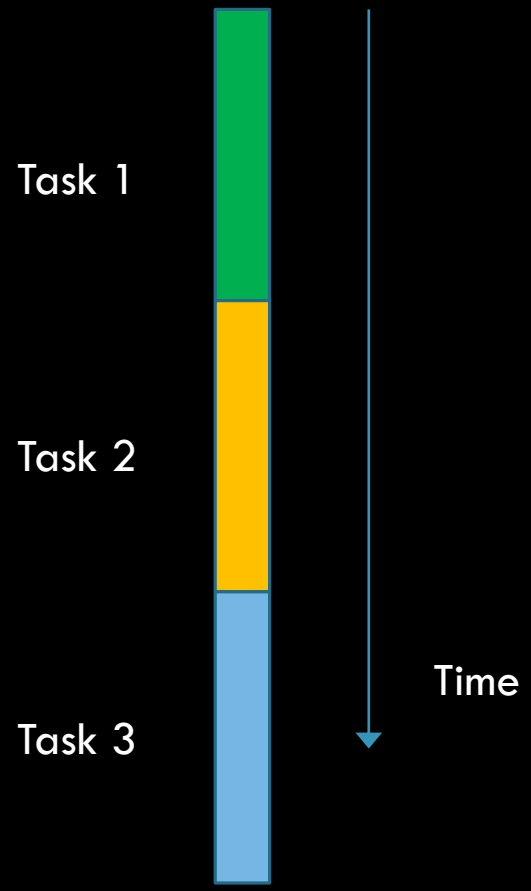
ASYNCHRONOUS MODEL

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

- It is computationally intensive and requires a certain amount of CPU time to calculate the answer.
- It is not computationally intensive but has to wait for data to be available to produce a result.

ASYNCHRONOUS MODEL

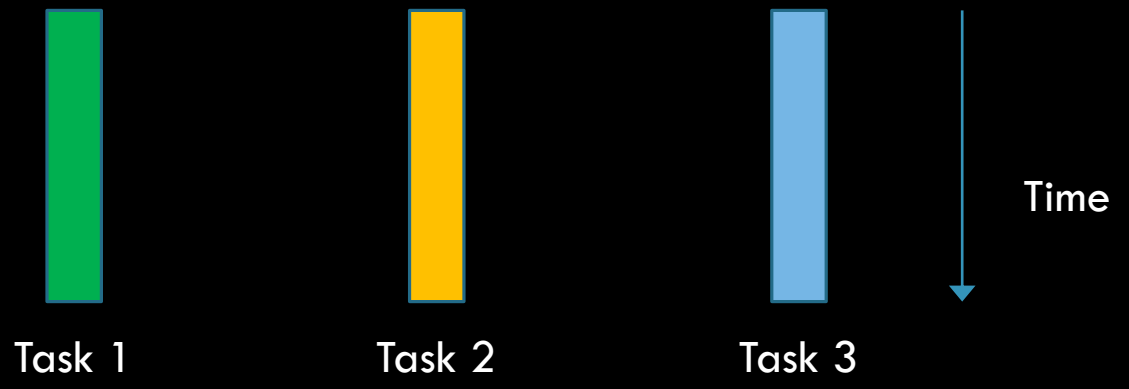
The simplest approach to the time-intensive task problem is to perform one task at a time, waiting until a task has finished before the next is started. This is known as the **synchronous** model.



ASYNCHRONOUS MODEL

The threaded model allows for complete tasks to be performed concurrently in separate threads.

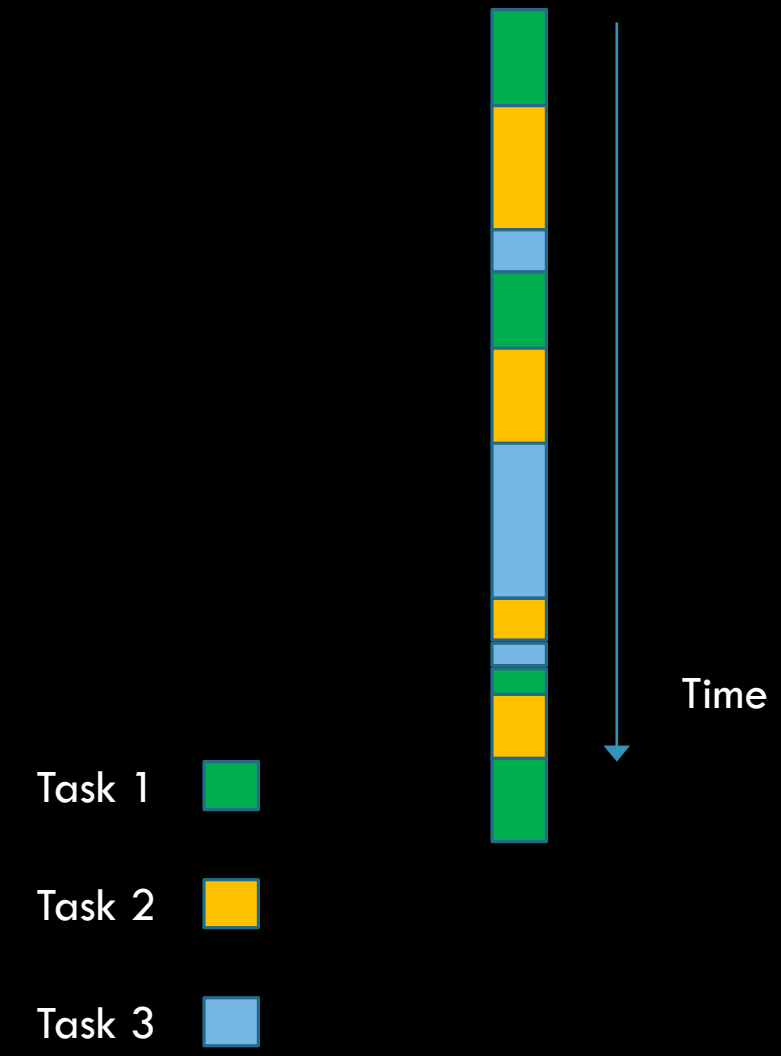
You should note that, when the resources are not available, threads will not truly run concurrently but, as the programmer, you can assume they do. The rest is handled by the OS.



ASYNCHRONOUS MODEL

In the **asynchronous** model, tasks are interleaved within a single thread of control. Only one task is being performed at a time, but it may relinquish control to another task before it has completed.

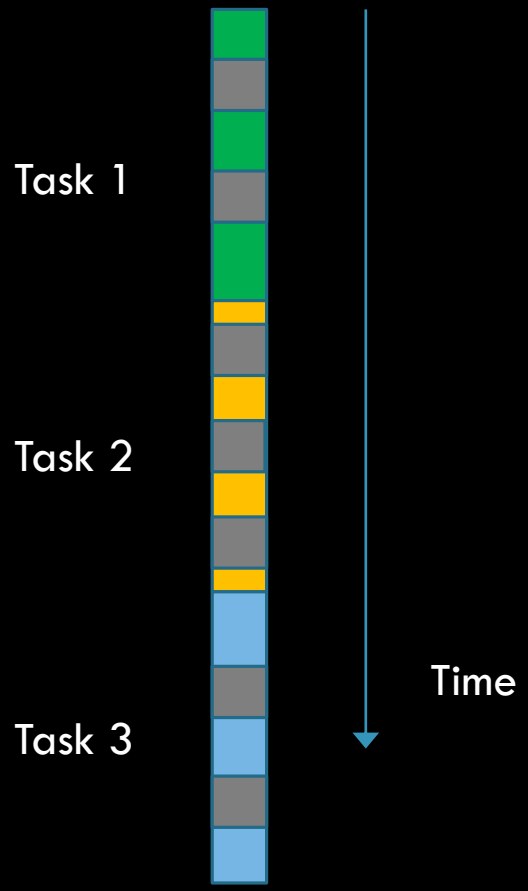
This is how threaded programs behave on single-processor systems. However, the difference is that threaded systems will look more like the second diagram when the resources are available. Single-thread asynchronous programs always interleave tasks.



ASYNCHRONOUS MODEL

One of the advantages of the asynchronous model is that it minimizes the time spent waiting on blocking actions.

Any time that would be spent waiting for some blocking call (external call or I/O, etc), we will use to make progress on another task within the same thread.



ASYNCHRONOUS MODEL

Compared to the synchronous model, the asynchronous model performs best when:

- There are a large number of tasks so there is likely always at least one task that can make progress.
- The tasks perform lots of I/O, causing a synchronous program to waste lots of time blocking when other tasks could be running.
- The tasks are largely independent from one another so there is little need for inter-task communication.

ASYNCHRONOUS MODEL

How does Twisted implement the asynchronous model? It enforces the use of non-blocking calls. The basic steps are these:

- A call is made using some function f. Let's say f is a scraper function, for example.
- The function f returns immediately with the expectation that a result will be returned when it is ready. That is, when the scraped results have been obtained, they will be returned.
- When the scraped results are obtained, the caller of f is notified via a *callback function* that the results are now available.

ASYNCHRONOUS MODEL

In synchronous programming, the thread will simply wait for any function call to completely finish.

In asynchronous programming, a function requests the data, and lets the library call the callback function when the data is ready.

To manage the callback sequence, Twisted uses *Deferred* objects. Deferreds have attached to them a sequence of callbacks and *errbacks* to be called when the results are finally returned.

If a function returns a Deferred object, we know that its task hasn't been completed – it's waiting on some data.

EVENT DRIVEN MODEL

Now, we know what it means to be asynchronous. What about event driven?

At the heart of any Twisted application is a *reactor loop*. The reactor loop, or event loop, is responsible for waiting on and dispatching events and/or messages within the application. This part is implemented by Twisted itself.

Basically, as a Twisted developer, you may develop functions and data to be used in a Twisted application, but the calling and scheduling of these functions is performed by the Twisted reactor itself.

TWISTED

Next class we'll learn about how to build a Twisted application and look at some examples.