LECTURE 10

Networking

NETWORKING IN PYTHON

Many Python applications include networking – the ability to communicate between multiple machines. We are going to turn our attention now to the many methods of network programming available in Python.

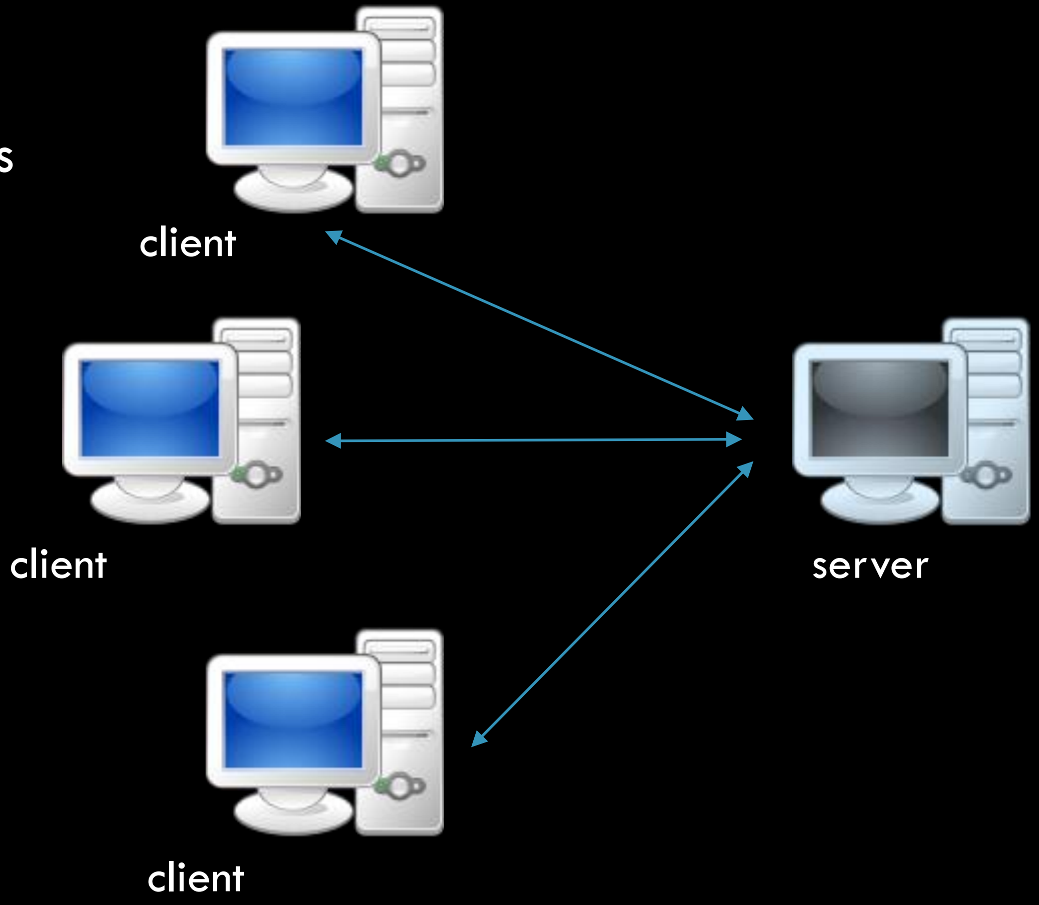
- Sockets
- SocketServer
- Twisted

But first, we must start with some networking fundamentals.

NETWORKING FUNDAMENTALS

In the **client-server** model, the client sends out a request to a server. The server processes the request and sends a response back to the client.

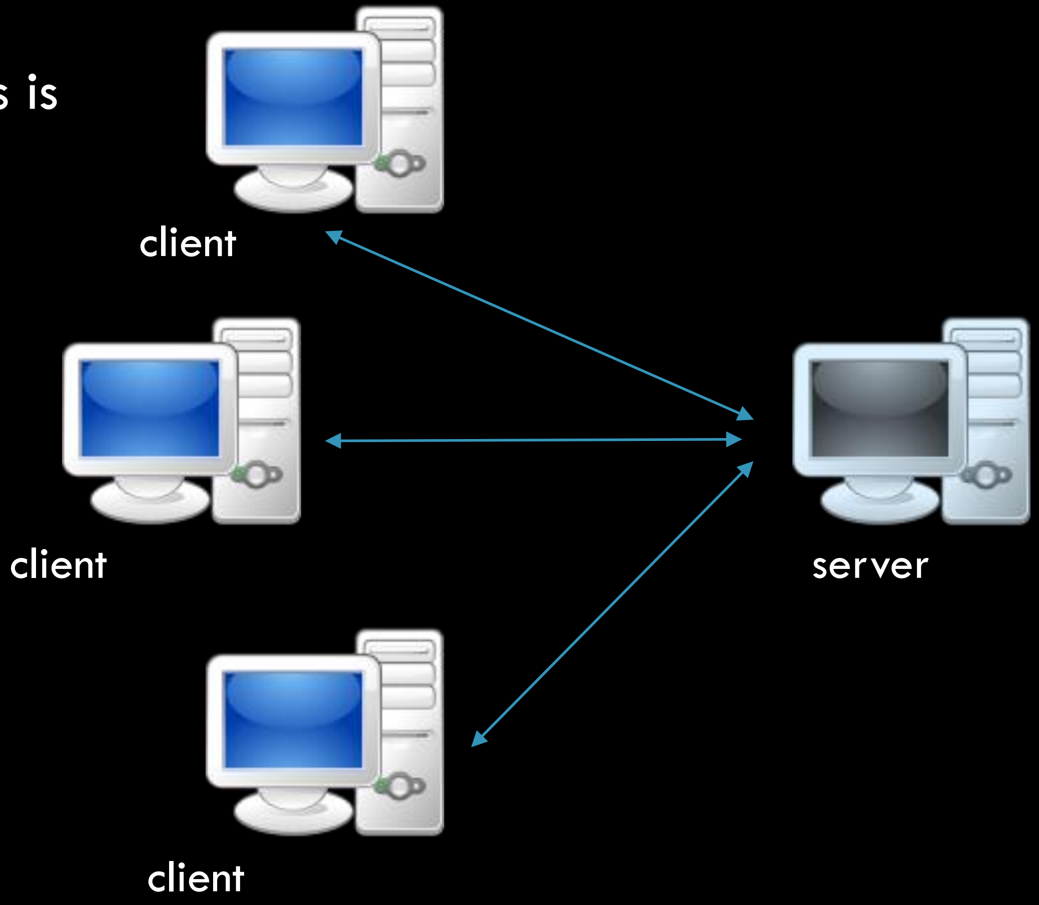
The classic example is a web browser sending a request for a webpage to a webserver. The webserver processes the request and returns the webpage to the browser.



NETWORKING FUNDAMENTALS

The coordination of the communication process is defined by the *protocol* being used.

For example, the web browser and webserver communication is defined by the TCP/IP communication protocol. This protocol defines how data should be packetized, addressed, transmitted, routed, and received over the Internet.

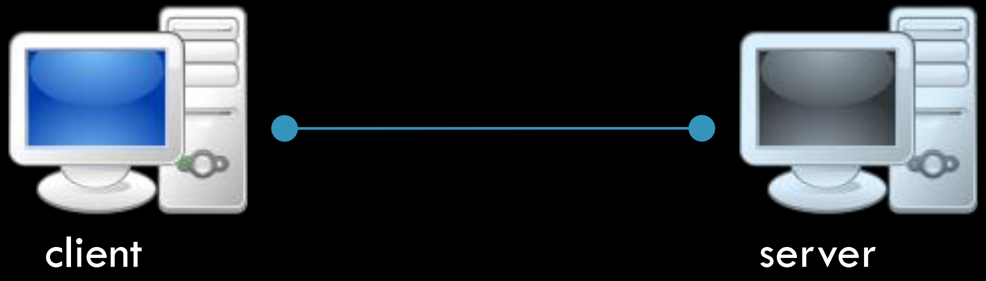


NETWORKING FUNDAMENTALS

A socket is simply an endpoint of communication.

They provide an abstraction for the details of communication. An application can interface with a socket object, which hides the details of the lower level network protocols.

Python has a socket module which provides access to the BSD socket interface. We will start with this simple, low-level way of creating a networked application.



THE SOCKET MODULE

- `socket.socket([`*`family`*`, [`*`type`*`]])` creates a socket object. Remember, this is one endpoint of a two-way communication link.
- The `family` argument is the address family. The default is `socket.AF_INET`, which tells the socket to support the IPv4 protocol (32-bit IP addresses). Other choices include:
  - `socket.AF_INET6` for IPv6 protocol (128-bit IP address).
  - `socket.AF_UNIX` for Unix Domain Sockets.
- The `type` argument is the type of socket.
  - `socket.SOCK_STREAM` for connection-oriented sockets (TCP).
  - `socket.SOCK_DGRAM` for datagram sockets (UDP).

In this class, we will only deal with TCP/IPv4 sockets. These are the default arguments.

THE SOCKET MODULE

So, let’s create our first socket.

```
from socket import *

s = socket(AF_INET, SOCK_STREAM) # or socket()
```

Now we have an IPv4 TCP socket which we can use to connect to other machines.

THE SOCKET MODULE

We've just created a socket object, which has a selection of standard methods available.

- `s.connect(`*`addr`*`)` creates a connection between the socket `s` and the specified address `addr`. The `addr` argument is a tuple containing the host name (as a string) and port number (as an int).
- `s.send(`*`string`*`)` sends a string to the address to which the socket is currently connected. The return value is the number of bytes sent. There is no guarantee that the whole message was sent – you should double check the return value!

THE SOCKET MODULE

The following code connects to the www.fsu.edu server and requests the page index.html. This is essentially what happens behind the scenes when you enter the address www.fsu.edu/index.html into your browser. The port number 80 is standard for http requests and the string we constructed is just a simple http GET request.

```python
from socket import *

s = socket()
s.connect(("www.fsu.edu", 80))
s.send("GET /index.html HTTP/1.0\n\n")
```

THE SOCKET MODULE

- `s.recv(`*`bufsize`*`)` receives and returns up to `bufsize` bytes of data from the address to which the socket is currently connected.
- `s.close()` closes the connection. Sockets are also automatically closed when garbage collected.

The `connect()`, `send()`, `recv()` and `close()` functions are all we need to write a simple TCP client. Clients simply create connections as necessary, send and receive data, and then close when they are finished with a transaction. Your browser is a client program – it connects to a server when it needs to and loses the connection when it's done.

THE SOCKET MODULE

Now, we’re not only sending a page request, but we receive 2048 bytes of data (in reality – the page is much bigger) and close the connection. The contents of data is simply 2048 bytes of the html of the page we requested.

```
from socket import *

s = socket()
s.connect(("www.fsu.edu", 80))
s.send("GET /index.html HTTP/1.0\n\n")
data = s.recv(2048)
s.close()
print data
```

THE SOCKET MODULE

To create a server, we can also use socket objects. We will just treat them as if they're the other endpoint of the connection.

- `s.bind(`*`addr`*`)` binds the socket object to an address `addr`. As before, `addr` is a tuple containing the hostname and port.
- `s.listen(`*`backlog`*`)` tells the socket to begin listening for connections. The `backlog` argument specifies how many connections to queue before connections become refused (default is zero).
- `socket.gethostname()` returns a string containing the local machine's primary public hostname.

THE SOCKET MODULE

Here we are creating a simple TCP server which is listening for connections on port 9000. If our server is already connected to another machine, we will queue up to 5 connections before we start refusing connections.

```
from socket import *

s = socket()
h = socket.gethostname()
s.bind((h, 9000))
s.listen(5)
```

THE SOCKET MODULE

Here we are creating a simple TCP server which is listening for connections on port 9000. If our server is already connected to another machine, we will queue up to 5 connections before we start refusing connections.

```
from socket import *

s = socket()
h = socket.gethostname()
s.bind((h, 9000))
s.listen(5)
```

We listen on a relatively high port number like 9000 because it is not reserved. Well-known services (HTTP, SFTP, etc) should listen on conventional port numbers.

We could also bind to localhost:

```
s.bind(("localhost", 9000))
s.bind(("127.0.0.1", 9000))
```

Or bind to any address the machine has:

```
s.bind(("", 9000))
```

THE SOCKET MODULE

- `s.accept()` passively waits until a connection is made. The return value is a pair `(conn, address)` where `conn` is a new socket object usable to send and receive data on the connection, and `address` is the address bound to the socket on the other end of the connection.

THE SOCKET MODULE

Here we're creating a simple TCP echo server.

The empty string argument to bind signifies that we're listening for connections to every hostname the machine happens to have.

Notice that we use a new, unique socket object c to communicate with the client.

We have a major practical problem however. What do you think it is?

```python
from socket import *

s = socket(AF_INET, SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print "Received connection from", a
    c.send("Hello " + str(a[0]))
    c.close()
```

THE SOCKET MODULE

- Clients
  - Connect when they need and close after a transaction.
  - `connect()`,`send()` and `recv()`,`close()`
- Servers
  - Listen for connections and communicate with each client using a new, unique socket object.
  - `bind()`,`listen()`,loop: `accept()`,`send()` and `recv()`,`close()`

Some useful utility functions:

- `socket.gethostbyname("www.python.org")` → returns IP
- `socket.gethostbyaddr("82.94.237.218")` → returns name

THE SOCKET MODULE

Sending and receiving data is often done in stages – these functions are bound to the capacity of the network buffers.

- `s.send(string)` returns the number of bytes sent.
- `s.recv(max)` may receive fewer bytes than the specified max. Returns an empty string when the connection is closed.
- `s.sendall(string)` blocks until all data has been transmitted. None is returned on success.

THE SOCKET MODULE

There are some additional options you can specify.

- `s.setblocking(`*`flag`*`)` uses `flag` to determine whether socket operations should be blocking or not. By default, all socket operations are blocking because they will wait for *something* to happen. For example, `recv()` will block until some data comes in.
- `s.settimeout(`*`t`*`)` sets a timeout of `t` seconds on all blocking socket operations. If `t` > 0, socket operations that exceed their time will raise a timeout exception. If `t` is `None`, the socket operation never times out.
- `s.makefile()` returns a file object which can be used to read and write to the socket object as a file object. VERY useful!

THE SOCKET MODULE

So, we've already mentioned that there's an issue with this code.

Practically speaking, this isn't a good server because it can only handle one connection at a time.

Incoming connections must wait for the previous connection request to be serviced before they are accepted. They might get refused outright, which is even worse.

```
from socket import *

s = socket(AF_INET, SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print "Received connection from", a
    c.send("Hello " + str(a[0]))
    c.close()
```

THE THREADING MODULE

To solve this one-connection-at-a-time issue, we'll introduce some concurrency tools.

The Python standard library includes a nice threading module which allows us to implement threading – running multiple operations concurrently within the same program space.

Simplest usage:

```
import threading
t = threading.thread(target=worker_function, args=(arg1,arg2))
```

Creates a thread `t` which will execute `worker_function` with arguments `arg1` and `arg2` when the thread is started.

THREADED TCP SERVER

```
import threading
from socket import *

def handle_client(c):
    ... whatever ...
    c.close()

s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    t = threading.Thread(target=handle_client, args=(c,))
    t.start()
```

This threaded version of a our simple TCP server can handle multiple connections at once.

As soon as a connection is accepted, the socket object is passed to a handling function in another thread. This allows the main thread to continue accepting new connections even when the first connection is still being served.

BLACKJACK_SERVER.PY

Let’s create a threaded TCP server which allows a client to play Blackjack.

Our little Blackjack game using the following simple rules:

- Dealer and Player both start with two cards in their hand. One of the dealer’s cards is hidden, the other is visible.
- The Player must try to add cards to their hand until they are as close to a combined value of 21 without going over. This is done by successively “hitting” until they are satisfied with their hand.
- Afterwards, the Dealer must add cards to his hand while the combined value is less than 17.
- The winner is the one whose cards have a combined total closest to 21 without going over.

TELNET TOOL

To test out your server applications, you do not need to build a client application to interface with it. Instead, use the telnet tool.

```
  $ telnet hostname port
```

will connect you to the server listening on hostname:port and you can interact with it as if you were a client application. For example,

```
  $ telnet localhost 9000
```

will allow you to connect and send information to a server listening locally on port 9000.