CSC361: Tutorial 1

Basics about python socket programming

Blocking and non-blocking sockets

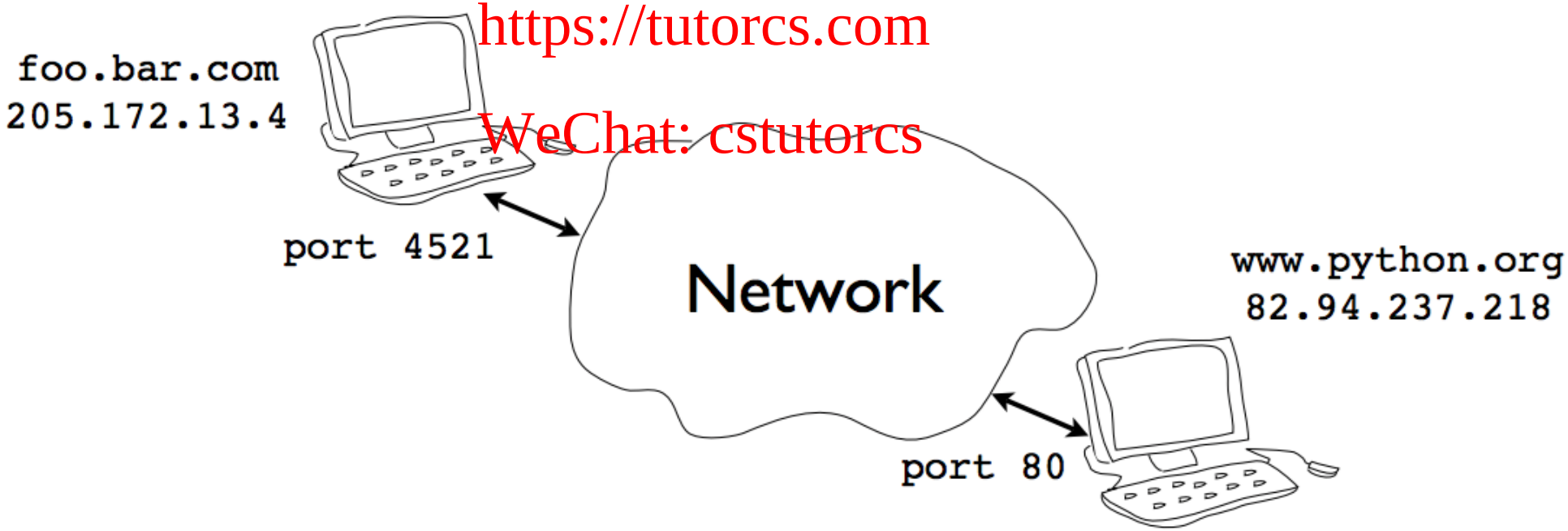
Intro to the select package

Download and Install Latest Python

- Download from https://www.python.org/downloads/

Network Addressing

- Machines have a hostname and IP address
- Programs/services have port numbers



- Each endpoint of a network connection is always represented by a host and port #

- In Python you write it out as a tuple (host,port)


```
("www.python.org",80)
```

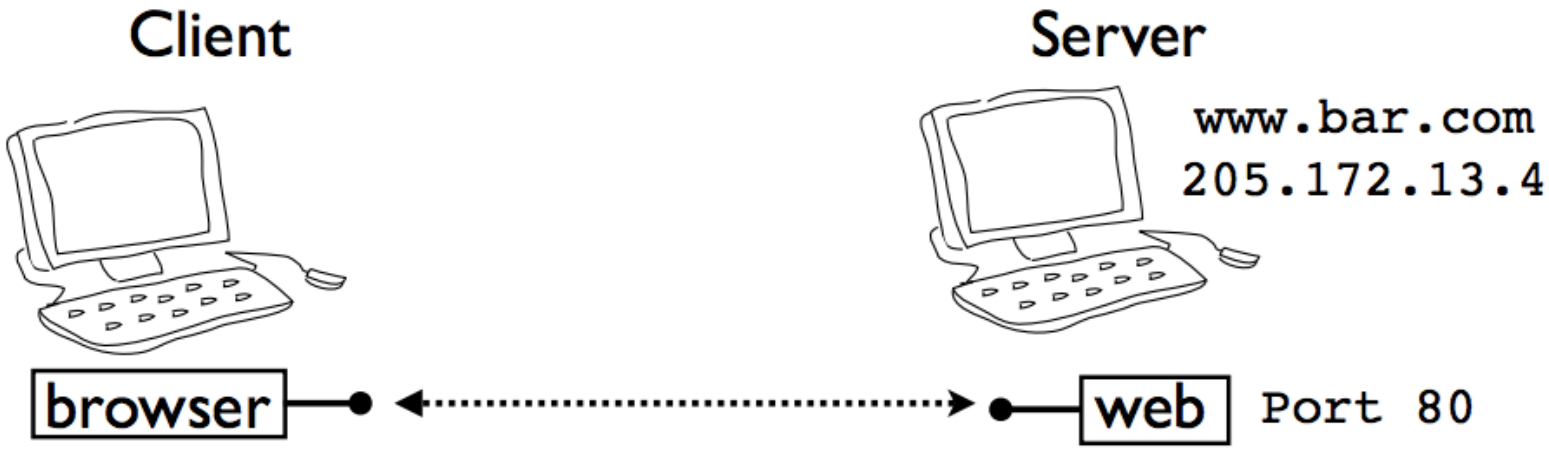


```
("205.172.13.4",443)
```


- In almost all of the network programs you'll write, you use this convention to specify a network address

Client/Server Concept

- Each endpoint is a running program
- Servers wait for incoming connections and provide a service (e.g., web, mail, etc.)
- Clients make connections to servers



- Client sends a request message (e.g., HTTP)

```
GET /index.html HTTP/1.0
```

- Server sends back a response message


```
HTTP/1.0 200 OK
Content-type: text/html
Content-length: 48823

<HTML>
...
```

Data Transport

- There are two basic types of communication
- Streams (TCP): Computers establish a connection with each other and read/write data in a continuous stream of bytes---like a file. This is the most common.
- Datagrams (UDP): Computers send discrete packets (or messages) to each other. Each packet contains a collection of bytes, but each packet is separate and self-contained.

Sockets

- Programming abstraction for network code
- Socket: A communication endpoint

- Allows connections to be made and data to be transmitted in either direction

Socket Basics

- To create a socket

```
import socket
s = socket.socket(addr_family, type)
```


- Address families

```
socket.AF_INET      Internet protocol (IPv4)
socket.AF_INET6     Internet protocol (IPv6)
```

- Socket types

```
socket.SOCK_STREAM   Connection based stream (TCP)
socket.SOCK_DGRAM    Datagrams (UDP)
```

Example: TCP Client

- How to make an outgoing connection

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.connect(("www.python.org",80))
s.send("GET /index.html HTTP/1.0\n\n")
data = s.recv(10000)
s.close()
```

Example: TCP Server

- Network servers are a bit more tricky
- Must listen for incoming connections on a port number
- Typically run forever in a server-loop
- May have to service multiple clients

- A simple server

```
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 %s\n" % a[0])
    c.close()
```


- Send a message back to a client

```
% telnet localhost 9000
Connected to localhost.
Escape character is '^]'.
Hello 127.0.0.1
Connection closed by foreign host.
%
```

Server message

- Address binding

```
from socket import *
s = socket(AF_INET,SOCK_STR
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print "Received connection from", a
    c.send("Hello %s\n" % a[0])
    c.close()
```

binds the socket to a specific address

- Addressing

```
s.bind(("",9000))
s.bind(("localhost",9000))
s.bind(("192.168.2.1",9000))
s.bind(("104.21.4.2",9000))
```

binds to localhost

If system has multiple IP addresses, can bind to a specific address

- Start listening for connections

```
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 %s\n" % a[0])
    c.close()
```

Tells operating system to start listening for connections on the socket

- s.listen(*backlog*)
- backlog is # of pending connections to allow
- Note: not related to max number of clients

- Accepting a new connection

```
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 %s\n" % a[0])
    c.close()
```

Accept a new client connection

- s.accept() blocks until connection received
- Server sleeps if nothing is happening

- Client socket and address

```
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 %s\n" % a[0])
    c.close()
```

Accept returns a pair (client_socket,addr)


```
<socket._socketobject
 object at 0x3be30>
```

This is a new socket that's used for data

```
("104.23.11.4",27743)
```

This is the network/port address of the client that connected

- Sending data

```
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 %s\n" % a[0])
    c.close()
```

Send data to client

Note: Use the client socket for transmitting data. The server socket is only used for accepting new connections.

- Closing the connection

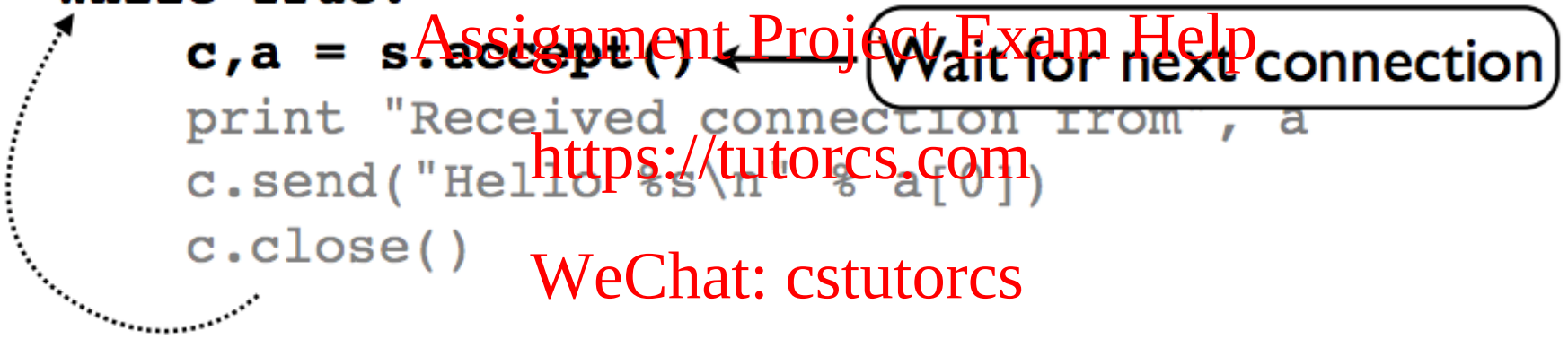
```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print "Received connection from", a
    c.send("Hello, world!\n")
    c.close()
```


Close client connection

- Note: Server can keep client connection alive as long as it wants
- Can repeatedly receive/send data

- Waiting for the next connection

```
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 %s\n" % a[0])
    c.close()
```



- Original server socket is reused to listen for more connections
- Server runs forever in a loop like this

The problem of the simple server

- It cannot serve multiple clients at the same time due to the blocking socket
- How to address?

Non-blocking sockets + the select package

Blocking and Non-Blocking Socket I/O

- By default, TCP sockets are placed in a **blocking mode**
  - A function is blocking if it has to wait for something to complete
  - For example, if you call the send() method, the process will transmit all the data to a buffer. When the buffer is full, the kernel will put the process to sleep until the data in the buffer is transferred to destination and the buffer is empty again.
- You can make a socket non-blocking by calling setblocking(0)
  - when you call the send() method, it will **write** as much data in the buffer as possible and return
  - If the buffer gets ***full*** and we continue to send data, socket.error will be raised.
  - When you try to send data more than the buffer can accommodate, only the amount of data that can be accommodated is actually sent and send() returns the number of bytes sent

```
import errno
import select
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('localhost', 1234))
sock.setblocking(0)

data = 'foobar\n' * 1024 * 1024
data_size = len(data)
print 'Bytes to send:', len(data)

total_sent = 0
while len(data):
    try:
        sent = sock.send(data)
        total_sent += sent
        data = data[sent:]
        print 'Sending data'
    except socket.error, e:
        if e.errno != errno.EAGAIN:
            raise e
        print 'Blocking with', len(data), 'remaining'
        select.select([], [sock], [])  # This blocks until

assert total_sent == data_size  # True
```

Set a non-blocking socket

Data exceeds the buffer

Use select to wait for the buffer can be written

Understanding select()

- The select module helps us with dealing with multiple file descriptors at once
- select() expects three arguments
  - list of file descriptors to watch for reading
  - list of file descriptors to watch for writing
  - list of file descriptors to watch for errors
  - Timeout can be passed as an optional 4th argument which can be used to prevent select() from blocking indefinitely
- select() returns a subset of all the three lists passed in the same order
  - i.e. all the file descriptors that are ready for reading, writing or have caused some error.

- In the above example, select() blocks if there is no file descriptor that is ready to work with

- As of now, select() will just block until our sock object becomes writeable again.

- If we remove that line, our script will continue to work but a lot more useless while loop iterations will be run as most of them will result in exceptions

- But how does select() really work? How we can use select() to build a server that can simultaneously serve multiple clients?

Example – echo server for multiple simultaneous connection

```
import select
import socket
import sys
import Queue

# Create a TCP/IP socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
 server.setblocking(0)

# Bind the socket to the port
server_address = ('localhost', 10000)
server.bind(server_address)

# Listen for incoming connections
server.listen(5)
```

- The next step in the server is to set up the lists containing input sources and output destinations to be passed to select().

```
# Sockets from which we expect to read
inputs = [ server ]
# Sockets to which we expect to write
outputs = [ ]
# Outgoing message queues (socket:Queue)
message_queues = {}
```

- The main portion of the server program loops, calling select() to block and wait for network activity.

```
while inputs:
# Wait for at least one of the sockets to be ready for processing
readable, writable, exceptional = select.select(inputs, outputs, inputs)
```


- select() returns three new lists, containing subsets of the contents of the lists passed in.
  - All of the sockets in the readable list have incoming data buffered and available to be read.
  - All of the sockets in the writable list have free space in their buffer and can be written to.
  - The sockets returned in exceptional have had an error

- The “readable” sockets have the following cases:
  - If the socket is the main “server” socket, the one being used to listen for connections, then the “readable” condition means it is ready to accept another incoming connection.

```
# Handle inputs
        for s in readable:
            if s is server:
                # A "readable" server socket is ready to accept a connection
                connection, client_address = s.accept()
                connection.setblocking(0)
                inputs.append(connection)
                # Give the connection a queue for data we want to send
                message_queues[connection] = Queue.Queue()
```


- We add the new connection to the list of inputs to monitor and set them as non-blocking sockets.

- The next case is an established connection with a client that has sent data. The data is read with recv(), then placed on the queue so it can be sent through the socket and back to the client.

```
else:
        data = s.recv(1024)
        if data:
                # A readable client socket has data
                message_queues[s].put(data)
                # Add output channel for response if s not in outputs:
                outputs.append(s)
        else:
                # Simply interpret empty result as closed connection
                # Stop listening for input on the connection if s in outputs:
                outputs.remove(s)
                inputs.remove(s)
                s.close()
                # Remove message queue
                del message_queues[s]
```

- There are fewer cases for the writable connections.
  - If there is data in the queue for a connection, the next message is sent.
  - Otherwise, the connection is removed from the list of output connections so that the next time through the loop select() does not indicate that the socket is ready to send data.

```
for s in writable:
    try:
        next_msg = message_queues[s].get_nowait()
    except Queue.Empty:
        # No messages waiting so stop checking for writability.
        outputs.remove(s)
    else:
        s.send(next_msg)
```

- Finally, if there is an error with a socket, it is closed.

```
# Handle "exceptional conditions”
        for s in exceptional:
                # Stop listening for input on the connection inputs.remove(s)
                if s in outputs:
                        outputs.remove(s)
                        s.close()
                        # Remove message queue
                        del message_queues[s]
```

Reference

- Brightspace -> Content -> t1: Python Network Programming.pdf

- Intro to select: https://pymotw.com/2/select/