# Assignment 3
# CS-471
# Spring 2018
**Due Date: 5/19/2018 at 11:59 pm (No extensions)**
**You may work in groups of 6**

## Goals:

1. **To understand** the challenges of protocol design.
2. **To discover** and appreciate the challenges of developing complex, real-world network applications.
3. **Make sense of** real-world sockets programming APIs.
4. **To utilize** a sockets programming API to construct simplified FTP server and client applications.

## Overview

In this assignment you will implement simplified versions of FTP server and client applications. The client shall connect to the server and support uploading and downloading of files to/from server. Before continuing **please review the lecture slides covering sockets programming and read the *Preliminaries* section, below**. Doing so will save you hours of frustration.

## The Preliminaries

In class we covered the basics of programming with TCP sockets. The purpose of this section is to give you further insight into the common pitfalls and caveats of the same. Reading and comprehending the material in this section will make this assignment **orders of magnitude** easier and more enjoyable.

### Peculiarities of send() and recv() and How to Get Them Right

Consider the following codes which implement simplified server and client programs, respectively:

```
# Server code
from socket import *

# The port on which to listen
serverPort = 12000

# Create a TCP socket
serverSocket = socket(AF_INET,SOCK_STREAM)

```

```
# Bind the socket to the port
serverSocket.bind(('',serverPort))

# Start listening for incoming connections
serverSocket.listen(1)

print "The server is ready to receive"

# The buffer to store the received data
data = ""

# Forever accept incoming connections
while 1:
        # Accept a connection; get client's socket
        connectionSocket, addr = serverSocket.accept()

        # Receive whatever the newly connected client has to send
        data = connectionSocket.recv(40)

        print data

        # Close the socket
        connectionSocket.close()
```

```
# Client code
from socket import *

# Name and port number of the server to
# which we are connecting.
serverName = "ecs.fullerton.edu"
serverPort = 12000

# Create a socket
clientSocket = socket(AF_INET, SOCK_STREAM)

# Connect to the server
clientSocket.connect((serverName,serverPort))

# A string we want to send to the server
data = "Hello world! This is a very long string."

# Send that string!
clientSocket.send(data)

# Close the socket
clientSocket.close()
```

In the code above, the client connects to the server and sends string `"Hello world!  This is a very long string."`. Both codes are syntactically correct. In theory, the server should print out the 40 character string sent by the client. In practice, this may not always happen. The potential problems are outlined below:

- **Problem 1: send(data) is not guaranteed to send *all* bytes of the string:** as we learned in class, each socket has an associated `send` buffer. The buffer stores the application data ready to be sent off. Whenever we call `send()`, behind the scenes, the operating system copies the data from the buffer in your program to the `send` buffer of the socket. The data will eventually be encapsulated into a TCP segment and handed over to the network layer.

  What happens if the send buffer does not have sufficient room to store the entire string? The answer is: `send()` will copy bytes until it fills the buffer and will return the number of bytes copied. This means that `send()` may fail to send the entire string.

- **Problem 2: data = connectionSocket.recv(40) is not guaranteed to receive *all* 40 bytes:** This can happen even if the sender has successfully sent all 40 bytes. Before delving into this problem, lets review the behavior of `recv()`. Function `recv()` will block until either a) the other side has closed its socket; or b) some data has been received. Case a) is generally not a problem. It can be easily detected by adding line `if not data:` immediately after `data = connectionSocket.recv(40)` in the receiver (which in the above example is the server). The test will evaluate to `True` if the client has closed its socket and to `False` otherwise. In Case b), the variable `data` will contain the received data. However, `data` may not be 40 bytes in size (again, even if the sender sent 40 bytes).

  Such behavior is possible because **not all data is guaranteed to arrive at the same time:** recall that Internet is a packet switched network. Big chunks of data are split into multiple packets. Some of these packets may arrive at the receiver faster than others. It is possible that, for example, the segment containing the first 20 bytes of the string arrives at the server, while the segment containing the last 20 bytes is still on its way. `recv()`, however, will return as soon as it gets the first 20 bytes.

So, what can we do about the above problems? The answer is that it is up to you, the application developer, to ensure that *all* bytes are sent and received. This can be done by calling `send()` and `recv()` functions inside the loop until all data has been sent/received. First, lets fix our client to ensure that it sends all of the specified bytes:

```
# Client code
from socket import *

# Name and port number of the server to
# which want to connect.
serverName = ecs.fullerton.edu
serverPort = 12000

# Create a socket
clientSocket = socket(AF_INET, SOCK_STREAM)

# Connect to the server
```

```
clientSocket.connect((serverName, serverPort))

# A string we want to send to the server
data = "Hello world! This is a very long string."

# bytesSent = 0

# Keep sending bytes until all bytes are sent
while bytesSent != len(data):
        # Send that string!
        bytesSent += clientSocket.send(data[bytesSent:])

# Close the socket
clientSocket.close()
```

We made three changes. First, we added an integer variable, `bytesSent`, which keeps track of how many bytes the client has sent. We then added line: `while bytesSent != len(data):`. This loop will repeatedly call `send()` until bytesSent is equal to the length of our data. Another words, this line says `"keep sending until all bytes of data have been sent."` Finally, inside the loop, we have line `bytesSent += clientSocket.send(data[bytesSent:])`. Recall, that `send()` returns the number of bytes it has just sent. Hence, whenever `send()` sends x bytes, `bytesSent` will be incremented by x. The parameter, `data[bytesSent:]`, will return all bytes that come after the first `bytesSent` bytes of `data`. This ensures that in the next iteration of the loop, we will resume sending at the offset of `data` where we left off.

With the client code working, lets now turn our attention to the server code. The modified code is given below:

```
# Server code
from socket import *

# The port on which to listen
serverPort = 12000

# Create a TCP socket
serverSocket = socket(AF_INET,SOCK_STREAM)

# Bind the socket to the port
serverSocket.bind(('',serverPort))

# Start listening for incoming connections
serverSocket.listen(1)

print "The server is ready to receive"

# Forever accept incoming connections
while 1:
        # Accept a connection; get client's socket
        connectionSocket, addr = serverSocket.accept()
```

```python

	# The temporary buffer
	tmpBuff = ""

	while len(data) != 40:

		# Receive whatever the client has to sent
		tmpBuff =  connectionSocket.recv(40)

		# The other side has unexpectedly closed it's socket
		if not tmpBuff:
			break

		# Save the data
		data += tmpBuff

	print data

	# Close the socket
	connectionSocket.close()
```

We made several changes. First, we added loop, `while len(data) != 40:`, which will spin until the size of `data` becomes 40. Hence, if `recv()` is unable to receive the expected 40 bytes after the first iteration, the loop will ensure that the program calls `recv()` again in order to receive the remaining bytes. Also, we changed line `data = connectionSocket.recv(40)` to `tmpBuff = connectionSocket.recv(40)` and added test `if not tmpBuff:` which will evaluate to `True` if the client has unexpectedly closed it's socket. We then added line `data += tmpBuff` which adds the newly received bytes to the data buffer. These changes ensure that with every iteration the newly received bytes are appended to the end of the buffer.

At this point we have a fully functioning server and client programs which do not suffer from the problems discussed above. However, there is still one important caveat: **in the above code, what if the server does not know the amount of data the client will be sending?** This is often the case in the real-world. The answer is that the client will have to tell the server. How? Well, this is where it is up to the programmer to design a communications protocol which allows the client to tell the serer just that.

One approach, for example, is to require that all messages sent from client to server start with a 10 byte header indicating the size of the data in the message. Hence, the server will always receive the first 10 bytes of message, parse them, determine the size of the coming data, and then use a loop as we did above, in order to receive the amount of data specified in the header. You can see such example in the directory `Assignment3SampleCodes/Python/sendfile`.

Before proceeding, it is recommended that you stop and think about what you just read, experiment with `Assignment3SampleCodes/Python/sendfile` codes, and make sure you understand everything.

## Specifications

Having mastered the preliminaries, you are now ready to start with design and implementation of your FTP client and server. This section outlines the requirements of both.

**The server shall be invoked as:** `python serv.py <PORT NUMBER>`

where `<PORT NUMBER>` specifies the port on which the FTP server will be accepting connection requests.

For example: `python serv.py 1234`

**The FTP client is invoked as:** `PYTHON cli.py <server_machine> <server_port>`

where `<server_machine>` is the domain name of the server (`ecs.fullerton.edu`). The domain name will be resolved to a 32-bit IP address using DNS. For example: `python cli.py ecs.fullerton.edu 1234`

Upon connecting to the server, the client prints out **ftp>**. Here the user can enter the following commands:

`ftp> get <file name>` (downloads file <file name> from the server)
`ftp> put <filename>` (uploads file <file name> to the server)
`ftp> ls` (lists files on the server)
`ftp> lls` (lists files on the client)
`ftp> quit` (disconnects from the server and exits)

***Use two connections for each FTP session - control and data***. Control connection lasts throughout the FTP session and is used exclusively to transfer all commands (i.e. ls, get, and put) from client to server and all status/error/information messages from server to client. This connection is established when the client first connects to the server.

Data channel is used exclusively for file data transfer. It is established and torn down for every file transfer – when the client wants to transfer data (ls, get, or put), the server generates an ephemeral port to use for data connection, notifies the client about the port on which it should connect, and then waits for the client to connect on that port. The connection is then used to upload/download file to/from the server. When the transfer is complete, the connection is closed.

When the client initially connects to the server (on the port specified at the command line), the server must print out the IP address and the port of the client.
For each command/request sent from the client to server (that is, all commands except lls), the server prints out a message indicating SUCCESS/FAILURE of the command. At the end of each transfer, client prints the filename and the number of bytes transferred.

# Designing the Protocol

Before you start coding, please design an application-layer protocol that meets the above specifications. Please submit the diagram and a detailed description of your protocol along with your code. Here are some guidelines to help you get started:

- What kinds of messages will be exchanged across the control connection?
- How does the other side respond to each type of message?
- What sizes/formats will the messages have?
- What message exchanges have to take place in order to setup a file transfer channel?
- How will the receiving side know when to start/stop receiving the file?
- You may want to use diagrams to model your protocol.

## Tips

- All sample files are in directory `Assignment3SampleCodes`. The files are as follows:
  - `ephemeral.py` illustrates how to generate an ephemeral port number. This program basically creates a socket and calls `bind()` in order to bind the socket to port 0; calling `bind()` with port 0 binds the socket to the first available port.
  - `cmds.py` which illustrates how to run an `ls` command from your code and capture its output.
  - Subdirectory `sendfile` contains files `sendfileserv.py` and `sendfilecli.py`, illustrating how to correctly transfer data over sockets (this includes large files).

## EXTRA CREDIT:

Implement the FTP client and server programs described above using a forking server and a threaded server.

## SUBMISSION GUIDELINES:

- This assignment may be completed using C++, Java, or Python.
- ***The assignment must be able to runnable on the Titan server or Topaz1 server (please contact me if you would like access to the latter).***
- Please hand in your source code electronically (do not submit .o or executable code) and protocol diagram and description through **TITANIUM**. You must make sure that your code compiles and runs properly.
- **Only one person within each group should submit.**
- Write a README file (text file, do not submit a .doc file) which contains
  - Names and email addresses of all partners.
  - The programming language you use (e.g. C++, Java, or Python)
  - How to execute your program.
  - Whether you implemented the extra credit.
  - Anything special about your submission that we should take note of.
- Place all your files under one directory with a unique name (such as `p3-[userid]` for assignment 3, e.g. `p3-mgofman1`).

- Tar the contents of this directory using the following command. `tar cvf [directory_name].tar [directory_name]` E.g. `tar -cvf p3-mgofman1.tar p3-mgofman1/`
- Use TITANIUM to upload the tared file you created above.

## Grading guideline:

- Protocol design 5'
- Program compiles: 5'
- Correct get command: 25'
- Correct put command: 25'
- Correct ls command: 10'
- Correct format: 10'
- Correct use of the two connections: 15'
- README file included: 5'
- BONUS: 15 points
- Late submissions shall be penalized 10%. No assignments shall be accepted after 24 hours.

## Academic Honesty:

**Academic Honesty:** All forms of cheating shall be treated with utmost seriousness. You may discuss the problems with other students, however, you must write your **OWN codes and solutions**. Discussing solutions to the problem is **NOT** acceptable (unless specified otherwise). Copying an assignment from another student or allowing another student to copy your work **may lead to an automatic F for this course**. Moss shall be used to detect plagiarism in programming assignments. If you have any questions about whether an act of collaboration may be treated as academic dishonesty, please consult the instructor before you collaborate. Details posted at http://www.fullerton.edu/senate/documents/PDF/300/UPS300-021.pdf.