Programming Assignment 1
CPSC-471, Spring 2024

Due Date: Saturday, 4/29/2024 at 11:59 pm

You may work in a group of at most 5 students.

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) FTP server and FTP client. The client shall connect to the server and support uploading and downloading of files to/from server. Before continuing, **please read the *Preliminaries* section, below**. It will save y o u hours of frustration.

The Preliminaries

In class we covered the basics of programming with TCP sockets. Please consult the slides covering the Application Layer, if you need a quick refresher. The purpose of this section is to list some tips which can make this assignment **orders of magnitude** easier and more enjoyable.

Peculiarities ofsend() and recv() and How to Get Them Right

Consider codes for client and server codes below.

```
# Server code
from s o c k e t import *

# The port on which to listen
server Po rt = 12000

# Create a TCP socke t
s e r v e r So c k e t = s o c k e t (AF INET ,SOCK STREAM)

# Bind the socket to the port
serv e rSo ck et . bind (( ' ' , serv er Po rt ))

# Start liste ning for incoming connections
serv er So cket . l i st e n ( 1 )

print "The s e r v e r i s ready to r e c e i v e "

# The buffer to s tore the received data
data = ""

# Forever accept incoming connections
while 1 :
# Accept a connection ; get client ' s socket
```

```
connection Socket , addr = serv er Socket . accept ()

# Receive whatever the newly connected client has to send
data = con nectio n Socket . recv ( 40 )
print data

# Close the socket
conn ection Socket . close ()
```

```
# Client code
from s o c k e t import ∗

# Name and port number of the serve r to
# which want to connect .
serverName = ” ecs . f u l l e r t o n . edu”
serv er Po rt = 12000
# Create a s ocke t
c li e nt S o c k e t = so ck et (AF INET, SOCKSTREAM)
# Connect to the serve r
cl i en t So ck e t . connect (( serverName , serv er Po rt ))
# A string we want to send to the serve r
data = ”Hel l o world ! This i s a very long st r i n g. ”
# Send that string!
c l i en t So ck e t . send ( data )

# Close the socket
c l i en t So ck e t . c lo s e ()
```

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. Whenever the client connects to the server, wewould expect the serverto print out the 40 characterstring sent by the client. In practice,thismaynot alwaysbethecase. Thepotentialproblems areoutlinedbelow:

- **Problem 1: send(data) is not guaranteed to send all bytes of the data:** As we learned in class, each socket has an associated send buffer. The buffer stores application data ready to be sent off. Whenever you call the send(), behind the scenes, the operating system copies the data from the buffer in your program into the send buffer of the socket. Your operating system will then take the data in the send buffer, convert it into a TCP segment, and push the segment down the protocol stack. send()**, however, will return immediately after the socket’s network buffer has been filled**. This means that in the client code above, if the data stringwas large, (i.e. Larger than the network buffer), the send() would return before sending all bytes of data.

- **Problem 2: data = connectionSocket.recv(40) is not guaranteed to receive all 40 bytes:** This can happen even if the sender has sent all 40 bytes. Before delving into this problem,letsreviewthebehaviorofrecv(). Functionrecv()willblockuntileither a) the other side has closed their 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 intheaboveexample is theserver). Thetestwillevaluatetotrueiftheothersidehas closed theirsocket.

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). The reasons can be as follows:

1. **Not all data may 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. Hence, it is possible that 20 bytes of the string arrive at the receiver, while 20 more are still on their way. recv(), however, will return as soon as it gets the first 20 bytes.

2. **recv() returns after emptying the receive buffer of the socket:** as we learned in class, each socket also has a receive buffer. That buffer stores all arriving data that is ready to be retrieved by the application. recv() **returns after emptying the receive buffer**. Because the receive buffer may not contain all data sent by the sender when the recv() is called, recv() will return before having received the specified numberof bytes.

So how do we cope with 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() inside the loop, until all data has been sent/received. First, lets fix our client to ensure that itsends all of the specified bytes:

```
# Clie n t code
from s o c k e t import *

# Name and port number of the serve r to
# which want to connect .
serverName = e c s . f u l l e r t o n . e d u
serv er Po r t = 12000

# Create a s ocke t
c l i e n t S o c k e t = s o c k e t (AF INET, SOCKSTREAM)

# Connect to the serve r
cl i en t So ck e t . connect (( serverName , serv er Po rt ))

# A string we want to send to the serve r
data = ” Hel l o world! This is a very long str i n g. ”

# byte sSent = 0

# Keep sending bytes until al l bytes are sent
while bytes Sent != len ( data ) :
# Send that s t r ing !
bytes Sent += c l i e n t S o c k e t . send ( data [ bytes Sent : ] )

# Close the s ocket
c l i en t So ck e t . c lo s e ()
```

Wemadethreechanges. First, weadded anintegervariable,bytesSent, whichkeepstrackof 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 havebeen sent." Finally, inside the loop, we have line bytesSent += clientSocket.send(data[bytesSent:]). Recall, that send() returns the number ofbytes 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 resumesendingattheoffsetofdatawhereweleftoff.

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

```
# Server code
from s o c k e t import ∗

# The port on which to listen
serv er Po r t = 12000

# Create a TCP s ocket
s e r v e r So c k e t = s o c k e t ( AF_INET ,SOCK_STREAM)

# Bind the s ocke t to the port
serv e rSo ck et . bind (( ’ ’ , serv er Po rt ))

# Sta rt l i s t ening for incoming connections
serv er So cket . l i st e n ( 1 )

print ”The s e r v e r i s ready to r e c e i v e ”

# Forever accept incoming connections
while 1 :
# Accept a connection ; get client ’s s ocke t
connection Socket , addr = server So cket . accept ()

# The temporary buffer
tmpBuff= ”” 
while len ( data ) != 4 0 :
# Receive whatever the newly connected client has to send
tmpBuff = con n ectio n Sock e t
.recv (40
# The other side u n expected ly clo s e d i t ’s s ocke t
if not tmpBuff :
break

# Save the data
data += tmpBuff

print data

# Close the s ocket
conn ection Socket . close ()
```

We made several changes. First, we added loop, while len(data) != 40:, which will spin untilthesizeofdatabecomes40. Hence,ifrecv()isunabletoreceivetheexpected 40 bytes afterthefirst call, theloop willensurethat theprogram calls recv() again inorder to receive the remaining bytes. Also, we changed line data = connectionSocket.recv(40)

totmpBuff= connectionSocket.recv(40) and added
test if not tmpBuff: which will evaluate to true if the other side unexpectedly closed it's socket. We then added line data += tmpBuff which adds the newly received bytes to the accumulator file 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 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 decide on the types and formats of messages that the server and client will be exchanging.

One approach, for example, is to decide that all messages sent from client to server start with a 10 byte header indicating the size of the data in the message followed by the actual data. Hence, the server will always receive the first 10 bytes of data, parse them and determine the size of the data, and then use a loop as we did above, in order to receive the amount of data indicated by the header. You can see such example in the directoryAssignment1SampleCodes/Python/sendfile.

Before proceeding, it is recommended that you stop and think about what you just read, experiment with Assignment1SampleCodes/Python/sendfile codes, and make sure you understand everything.

Specifications

The server shall be invoked as:

pythonserv.py<PORTNUMBER>

<PORT NUMBER>specifiestheport atwhichftp serveraccepts connection requests.
For example: python serv.py 1234

The ftp client is invoked as:

cli <server machine> <server port>

<server machine> is the domain name of the server (ecs.fullerton.edu). This will be converted into 32 bit IP address using DNS lookup. For example: python cli.py ecs.fullerton.edu 1234

Upon connecting to the server, the client prints out **ftp>**, which allows the user to execute 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 theserver)
ftp> quit (disconnects from the server and exits)

Use two connections for each ftp session - control and data. Control channel lasts throughout the ftp session and is used to transfer all commands (ls, get, and put) from client to server and all status/error messages from server to client. The initial channel on which the client connects to serverwill be the control channel. Data channel is used for data transfer. It is established and torn down for every file transfer – when the client wants to transfer data (ls, get, or put), it generates an ephemeral port to use for connection and then wait for the server to connect to the client on that port. The connection is then used to upload/download file to/from the server, and is torn down after the

transfer is complete.

For each command/request sent from the client to server, the server prints out the message indicating SUCCESS/FAILURE of the command. At the end of each transfer, client prints the filenameandnumber of bytestransferred.

Designing the Protocol

Before you start coding, please design an application-layer protocol that meets the above specifications. Please submit the design along with your code. Here are some guidelines to help you get started: • What kinds of messages will be exchanged across the control channel?
- How should the otherside respond to the messages?
- Whatsizes/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?
- How to avoid overflowing TCP buffers?
- You may want to use diagrams to model your protocol.

Tips

- All sample files are in directory Assignment1SampleCodes. The following files contain sample Python codes, should you choose to use this language.
  - Please see sample file, ephemeral.py, illustrating how to generate an ephemeral portnumber. This program basically creates a socket and calls bind() in order to bind the socket to port0; calling bind() with port 0 binds the socket to the first available port.
  - Please see file cmds.py which illustrates how to run an ls command from your code and to capture itsinput.
  - Subdirectory sendfile contains files sendfileserv.py and sendfilecli.py, illustrating how tocorrectlytransferdataoversockets(thisincludeslargefiles).

SUBMISSIONGUIDELINES:

- This assignment may be completed using C++, Java, or Python.
- Please hand in your source code electronically (do not submit .o or executable code) through **CANVAS**. You must make sure that this code compiles and runs correctly.
- **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.
  - Anything special about your submission that we should take note of.
- Place all your files under one directory with a unique name (such as p1-[userid] for assignment 1, e.g. p1-ytian1).
- Tarthe contents of this directory using the following command(Linux).tar cvf [directory_ name].tar [directory_name] E.g. tar -cvf p1-ytian1.tar p1- ytian1/
- Use CANVAS to upload the tared file you created above.

Grading Guideline:

- **Protocol design 10'**
- Program compiles: 5'
- Correct get command: 25'
- Correct put command: 25'
- Correct ls command: 10'
- Correct format: 5'
- Correct use of the two connections: 15'
- README file included: 5'
- Late submissionsshall be penalized 10%. No assignmentsshall 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 w r i t e 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.