

CIS 457 - Winter 2017

Project 1 - TCP File Transfer

February 8, 2017

Ellysa Stanton
Grand Valley State University
stantoel@mail.gvsu.edu

Adam M. Terwilliger
Grand Valley State University
terwilli@mail.gvsu.edu

Objective

Become experienced with stream (TCP) sockets though implementing a program.

Deliverables

You must turn in your code on blackboard on the part two due date. Additionally, you must turn in your documentation in by the next lecture after the part two due date. You must demo your client and server meeting requirements for each part on or before their respective due dates (according to the date of your enrolled lab section). Lab time may be available for these demos, but if there is not sufficient time, you are responsible for arranging a time to complete the demo.

Introduction

In this project, you must implement a file transfer client and server. You must write this program in Java, C, or C++, or Python (other languages may be allowed by request). You may not use any non-standard libraries without prior permission. Network communication must be done using stream (TCP) sockets. Your programs must run on the datacomm lab linux computers.

Part 1

In part 1, you will implement a client/server which transfer single file. Both client and server may exit after the transfer or keep running, it is up to you. There is no need in this part for the server to support multiple clients.

The file written by the client should be an exact match to the file sent. You can verify this with diff. Files that are approximate matches will get partial credit, but do not rely on a file working properly as an indication that it transferred correctly. Many programs will open files with minor errors.

For part 1 your server must do the following:

1. When started, your server must wait for a connection from a client on a certain port number (specified any way you would like, but not hard coded). Be sure to return an error if the port specified is not valid to use.
2. When your server gets a connection from a client, it must print a message to screen indicating a client connected.
3. When receiving a message indicating the client has specified a file to transfer, the server must read that file from disk if it exists and send it to the client. If the file does not exist, the server must indicate this to the client.
4. For part 1, your server is only required to support a single client at a time, there is no need to support multiple simultaneous clients.
5. The server must continue running after the client exits. Multiple clients in sequence must be supported.

Your client must do the following:

1. Upon startup, you client should ask the user for an IP address and port number of a server. This input should be checked for validity. It must then connect to the server at the address indicated, displaying an error and exiting if the server is not reachable, or if the user input is not a valid address and port.
2. After connecting to the server, the client should send a request for a file name specified by the user. After sending the request the client should receive the file contents from the server and save them to disk.

Part 1 is worth 45 points, divided as follows:

Criteria	Points
Server waits for client connection	5
User specified server port	5
Client connect to server	5
Client sends request message	5
File reading, sending, writing works for text files	10
File reading, sending, writing works for all files	10
Server continues running after client exits	5

Part 2

In part 2, you must extend part 1 by adding some extra functions a user normally expects from a file transfer program:

1. You must now check for errors in all user input, and notify the user.
2. No error should cause your server to exit. Instead, the user should be given an opportunity to re-enter their input, until they enter something valid.
3. You must allow clients to request a list of files available on the server.
4. Your client and server must also support multiple file transfers. The client may support multiple transfers sequentially, however the server must support multiple clients transfers simultaneously. Specifically, the following functionality related to multiple file transfers is required:
5. The client must no longer disconnect after a single file transfer. Instead, it should stay connected to the server and ask the user to either specify a new file or to exit.
6. If a new file is specified, it must be requested by the client over the same connection used to request and receive the previous file.
7. If the user chooses to exit, both the client and server must cleanly shut down their connections. The server should print a message when the client disconnects.
8. The server must allow multiple simultaneous client connections and be able to send a file to each at the same time.

Part 2 is worth 45 points, divided as follows:

Criteria	Points
All user input checked for potential errors	5
Errors handled in appropriate ways	5
File list available from the server	5
Multiple files transferred by same client	5
Same TCP connection used for multiple files	10
Clean disconnect on client exit	5
Server handles multiple clients simultaneously	10

Documentation

Documentation of your program is worth 10 points. This should be a 2-4 page document describing the design of your program. This should not be a line by line explanation of your code. It should detail any major design decisions and alternatives considered. For this project an example of such a design decision is the method used to determine when a file is done transferring.

Implementation

Summary

As per the project requirements, we designed a multi-threaded server that can serve multiple clients simultaneously for file transfers. The choice of programming language was Python, due to comfortability and ease of design.

Threading

It should be noted that due to the Global Interpreter Lock in Python, true parallelism using multi-threading was not available. However, we utilized the Python threading library (multiple threads running on same core) to demonstrate understanding through a pedagogical process. Despite the fact that Python is an interpreted language and is known for being an easily readable and writable language; using the threading library requires a bit of an object-oriented approach using class structure and constructors. This can be seen in the following client code snippet:

```
# using Python threads for pedagogical purposes
# as due to the global interpreter lock
# Python is not truly "multi-threaded"
class myThread (threading.Thread):

    # initialize thread class
    # pass socket and filename as parameters
    def __init__(self, s, filename):
        threading.Thread.__init__(self)
        self.s = s
        self.filename = filename

    # run script executes on thread.start()
    def run(self):
```

We note that an `__init__` constructor is needed. Additionally, similar to an in-case code example in Java, a `run` method is needed within the threading class. A thread can be instantiated and started easily as seen below. This code also is infinitely looping, and hence can serve multiple clients (threads) for file transfers. The global variable, `clientcount`, serves as a quit condition within the thread, notifying the server when no clients are still connected.

```
while 1:
    s.listen(1)
    conn, addr = s.accept()
    print 'Connected by', addr
    clientcount += 1
    thread = myThread(conn)
    thread.start()
```

File Transfer

File transfer is successful and functional in this implementation. However, the method may not be as efficient as desired, and serves as future work. The client sends a filename string to the server. If found, the server sends a file length string, so the client knows what size packet to expect the file. Then the server sends the file.

We noted improvements could be made, and in two ways: chunking for big files and not appending blanks to the original file length string sent from the server to the client. We can see this process in the following server code snippet:

```
# read this file
myfile = open(filename, 'r')
myfilestr = myfile.read()

myfilelen = str(len(myfilestr))

# so client knows size of file length packet
# append blanks on end of file length string
mynewfilelen = appendBlanks(myfilelen,100)

# send file length string first
self.conn.sendall(mynewfilelen)

# then send file
self.conn.sendall(myfilestr)
```

Error Handling

Errors are handled in many instances throughout this program. The first instance is with user input of host-name and port numbers. We utilize a regular expression obtained from stack overflow for host-names and implement our own port number check. This regular expression and the implementation of the error handling can be seen below:

```
#Regular Expression check for valid host name
#http://stackoverflow.com/questions/2532053/validate-a-hostname-string
def is_valid_hostname(hostname):
    if len(hostname) > 255:
        return False
    if hostname[-1] == ".":
        hostname = hostname[:-1] # strip exactly one dot from the right, if present
    allowed = re.compile("(?!-)[A-Z\d-]{1,63}(?!\-)$", re.IGNORECASE)
    return all(allowed.match(x) for x in hostname.split("."))

def main():
    host = raw_input('host: ')

    # use regular expression function
    while not is_valid_hostname(host):
        print "Invalid host, try again."
        host = raw_input('host: ')

    port = raw_input('port: ')

    # check for non-characters and negative or large port numbers
    while not port.isdigit() or int(port) < 0 or int(port) > 65536:
        print "Invalid Port Number, try again."
        port = raw_input('port: ')
```

The next level of error checking is done on the server side with regards to proper filenames. The client can request a list of files (ls), a valid file in the server directory (local directory), and has the option to quit. The server obtains a list of available files during instantiation and checks this list for validity of the requested file from the client. The implementation of this error checking can be seen below:

```
# list of files in current directory
def __init__(self, conn):
    threading.Thread.__init__(self)
    self.conn = conn
    self.serverfl = os.listdir(".")

def run(self):
    global clientcount

    # allow client to request as many files as desired
    while 1:

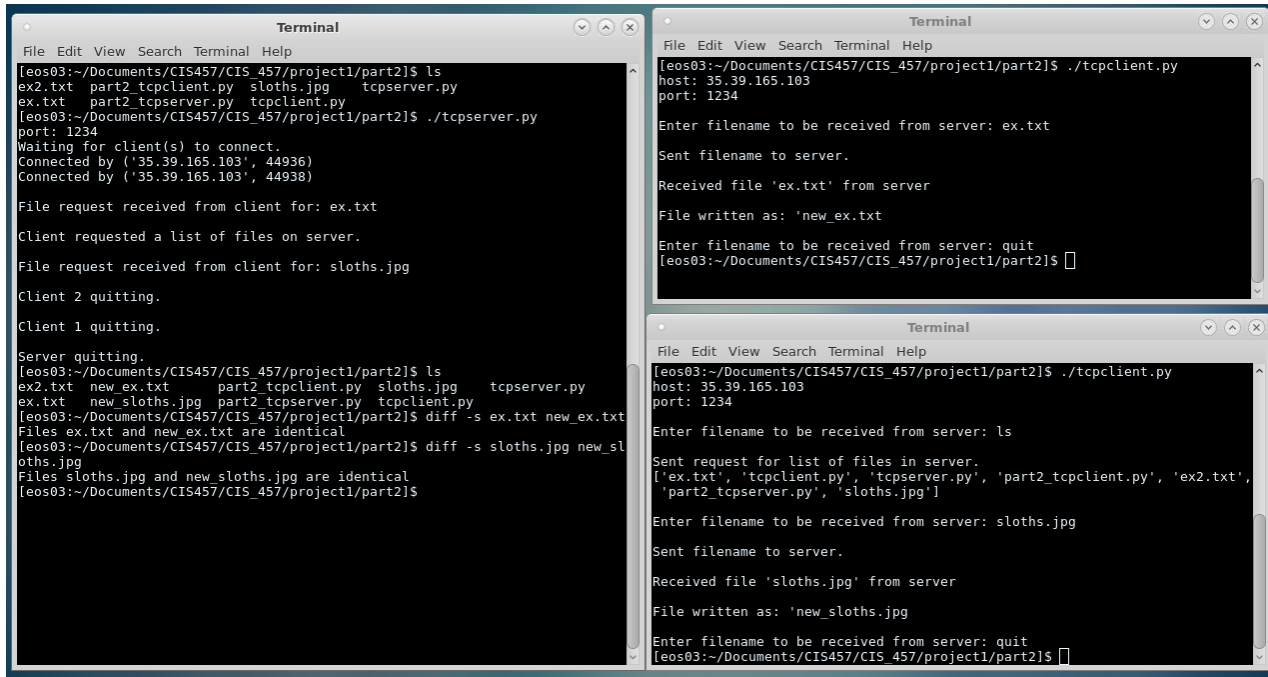
        # receive filename from client
        filename = self.conn.recv(1024)

        # only quit when no clients are still connected
        if filename == "Client quitting.":
            print ("\nClient {} quitting.".format(clientcount))
            clientcount -= 1
            if (clientcount == 0):
                print "\nServer quitting."
                self.conn.close()
                os._exit(1)

        # send list of files casted as string
        elif filename == "ls":
            print "\nClient requested a list of files on server."
            self.conn.sendall(str(self.serverfl))

        # if file is in list of files, go ahead and send
        elif filename in self.serverfl:
```

Sample Output



```
Terminal
File Edit View Search Terminal Help
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$ ls
ex2.txt  part2_tcpclient.py  sloths.jpg  tcpserver.py
ex.txt   part2_tcpserver.py  tcpclient.py
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$ ./tcpserver.py
port: 1234
Waiting for client(s) to connect.
Connected by ('35.39.165.103', 44936)
Connected by ('35.39.165.103', 44938)

File request received from client for: ex.txt

Client requested a list of files on server.

File request received from client for: sloths.jpg

Client 2 quitting.

Client 1 quitting.

Server quitting.
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$ ls
ex2.txt  new_ex.txt  part2_tcpclient.py  sloths.jpg  tcpserver.py
ex.txt   new_sloths.jpg  part2_tcpserver.py  tcpclient.py
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$ diff -s ex.txt new_ex.txt
Files ex.txt and new_ex.txt are identical
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$ diff -s sloths.jpg new_sloths.jpg
Files sloths.jpg and new_sloths.jpg are identical
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$
```

```
Terminal
File Edit View Search Terminal Help
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$ ./tcpclient.py
host: 35.39.165.103
port: 1234

Enter filename to be received from server: ex.txt

Sent filename to server.

Received file 'ex.txt' from server

File written as: 'new_ex.txt'

Enter filename to be received from server: quit
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$
```

```
Terminal
File Edit View Search Terminal Help
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$ ./tcpclient.py
host: 35.39.165.103
port: 1234

Enter filename to be received from server: ls

Sent request for list of files in server.
['ex.txt', 'tcpclient.py', 'tcpserver.py', 'part2_tcpclient.py', 'ex2.txt',
'part2_tcpserver.py', 'sloths.jpg']

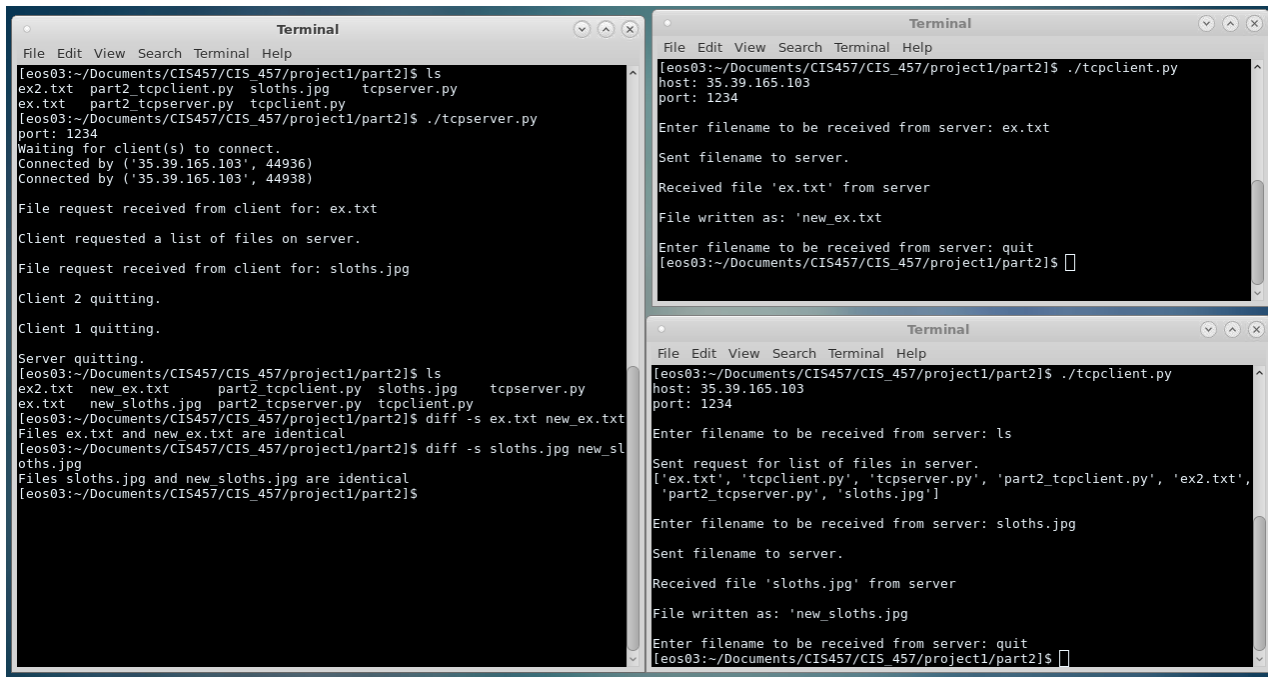
Enter filename to be received from server: sloths.jpg

Sent filename to server.

Received file 'sloths.jpg' from server

File written as: 'new_sloths.jpg'

Enter filename to be received from server: quit
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$
```



```
Terminal
File Edit View Search Terminal Help
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$ ls
ex2.txt  part2_tcpclient.py  sloths.jpg  tcpserver.py
ex.txt   part2_tcpserver.py  tcpclient.py
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$ ./tcpserver.py
port: 1234
Waiting for client(s) to connect.
Connected by ('35.39.165.103', 44936)
Connected by ('35.39.165.103', 44938)

File request received from client for: ex.txt

Client requested a list of files on server.

File request received from client for: sloths.jpg

Client 2 quitting.

Client 1 quitting.

Server quitting.
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$ ls
ex2.txt  new_ex.txt  part2_tcpclient.py  sloths.jpg  tcpserver.py
ex.txt   new_sloths.jpg  part2_tcpserver.py  tcpclient.py
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$ diff -s ex.txt new_ex.txt
Files ex.txt and new_ex.txt are identical
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$ diff -s sloths.jpg new_sloths.jpg
Files sloths.jpg and new_sloths.jpg are identical
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$
```

```
Terminal
File Edit View Search Terminal Help
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$ ./tcpclient.py
host: 35.39.165.103
port: 1234

Enter filename to be received from server: ex.txt

Sent filename to server.

Received file 'ex.txt' from server

File written as: 'new_ex.txt'

Enter filename to be received from server: quit
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$
```

```
Terminal
File Edit View Search Terminal Help
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$ ./tcpclient.py
host: 35.39.165.103
port: 1234

Enter filename to be received from server: ls

Sent request for list of files in server.
['ex.txt', 'tcpclient.py', 'tcpserver.py', 'part2_tcpclient.py', 'ex2.txt',
'part2_tcpserver.py', 'sloths.jpg']

Enter filename to be received from server: sloths.jpg

Sent filename to server.

Received file 'sloths.jpg' from server

File written as: 'new_sloths.jpg'

Enter filename to be received from server: quit
[leos03:~/Documents/CIS457/CIS_457/project1/part2]$
```