Programming Assignment 1: Socket Programming V1.1

Assigned 9/17/20
Due: 9/24/20

In this assignment, you’ll write separate, simple client and server programs that will communicate with each other using sockets, according to the protocol described below. You’ll first write the client and server using UDP sockets to communicate, and then write another version of the client and server using TCP sockets to communicate.

Here is the protocol. It’s a simple one – mostly to allow you to write a client and server that communicate using sockets – but we’ll use something similar (the TCP version) as a building block in the next assignment.

Client: (initiator) sends a message to the server with the string “HELLO” followed by a space, followed by a string containing a random 4-digit number that we will refer to as the connection ID. So, the initial client string might look like:

- HELLO ConnectionID
  for example, HELLO 2876

How to structure and run the client

The client’s input string is a console (command-line) argument. For example, if your client is in python, you need to run it in your console (command-line) as below.

- python your_client.py HELLO Server_IP Server_Port ConnectionID
  for example, python UDP_client.py HELLO 192.168.0.15 34567 2876

Anyone should be able to run your client to connect to different servers on different ports or connection IDs via the console without looking at your code. So, the entire message needs to be a console (command-line) argument.

Server. The server opens a port on a specific port number that you choose and waits for a client to connect at the listening port. The server receives the client’s message via a TCP segment socket or a UDP datagram socket (you’ll program both, separately). Then, the server application

---

1 V1.0-to-V1.1: clarified client command-line input on page 1.
reads the client’s message and responds with one of the two responses. The server keeps a list of in-use connection IDs.

- **OK.** If the connection ID specified by the client is not in use, it adds the connection ID to its list of in-use IDs and responds with an OK message. The OK message starts with the string “OK” and echoes back the client-provided connection ID. Also, the server attaches the client IP address and client port number to the return message. Thus, the server message looks like “OK Connection_ID Client_IP Client_Port” For example, if the client IP address is 192.168.0.10, the client port is 12345, and the connection ID is 9876, the return message from the server to the client would be “OK 9876 192.168.0.10 12345” [Note: no quotation marks in the string itself. We’re using quotation marks in the writeup to delimit strings, as needed].

- **Error.** If the connection ID specified by the client is in use, it responds with a RESET message. The RESET message starts with the string “RESET” and echoes back the client provided connection ID. In the case of TCP, the server will continue to wait for other clients’ requests at the welcoming (listening) socket but will close the connection/socket opened by the client that requested this connection. In the UDP case, there is no connection to close, and the server will again wait for another UDP datagram. To sum up, the server message is “RESET Connection_ID”. For example, if the client IP address is 192.168.0.10, the client port is 12345, and the connection ID is 9876, the return message from the server to the client would be “RESET 9876”. [Note: no quotation marks in your string itself. We’re using quotation marks in the writeup to delimit strings, as needed.]

- **Timeouts.** There are two timeout events you need to consider:
  - **Server Timeout:** when a server (whether TCP or UDP) is waiting for a connection request but does not receive any request for **five minutes** should timeout and exit gracefully after closing any open sockets. You will need to familiarize yourself with Python timer objects to implement this (initialize, start, restart timers, handle timer interrupts). You’ll use timers in the next assignment also. So get to know how to use them now.
  - **Connection_ID Timeout:** the server should timeout and remove connection IDs that have been in its connection ID list for more than **1 minute**.

In summary, the steps performed by the server are:

1. Open a socket on a specific port as a server and
2. Listen to the socket (if TCP) or read from that socket (if UDP)
3. Receive a request which consists of a HELLO and a connection ID.
4. Check the connection ID and return the appropriate response to the client regarding the protocol described above.
5. Handle timeouts every minute and gracefully exit the server if there have been no requests in 5 minutes.
How to run the server

Like the client, you must specify the server IP address and port number as inputs when you start the server from the console (command line). For example, a console command to run a server written in python3 would be:

```
python3 UDP_server.py Server_IP Server_Port
```

In this way, anyone could run your server code on an arbitrary server using an arbitrary port number without touching your code and changing the variables.

Client (final step)

- If the client receives an OK message, it prints out a “Connection established” message, indicating the connection ID, its IP address, and its port number. Then, the client exits gracefully after closing any open sockets.
  - OK print out example: “Connection established 2876 192.168.0.10 12345”

- Suppose the client receives a RESET message or doesn’t receive a reply within 60 seconds. In that case, it prints out a “Connection Error” message, indicating the connection ID, and asks the user to enter a new connection ID. Then, it retries to establish a connection again with this newly-entered connection ID. After three tries to establish a connection without success, the client prints a “Connection Failure” message and exits gracefully after closing any open sockets.
  - Connection Error print out example: “Connection Error 2678”
  - After three tries, print out: “Connection Failure”

In summary, the steps for the client are as follows:

1. Read HELLO ServerIP ServerPort Connection_ID message from the console.
2. Open a socket (TCP or UDP) to the server using the given ServerIP and ServerPort provided in the command line input.
3. Send “HELLO Connection_ID” as a message to the server, and wait for the results back.
4. Read the result from the server.
5. If the result is OK, display the print out a “Connection established Connection_ID Client_IP Client_Port” message. For an ERROR message or 60 seconds timeout, follow the description above. [Note: quotation marks used to delimit strings].

[Note: please do not print extra information or logs in the output. You could print any information or log to debug your code but remember to comment them before submitting your code.]
Getting Started: advice

Assuming this is your first experience with socket programming, we strongly recommend implementing the most straightforward client and server program first. For example, try UDP sockets, and forget initially about the timeouts. Have the client and server both running on your own same computer (see Part 1 below). Get them to talk with each other. Then try putting in additional functionality, piece by piece. Then tackle the TCP client and server.

Part 1

You’ll first develop these programs on your own computer, using the so-called “loopback interface” to connect the client socket to the server socket. With a loopback interface, your client and server will be separate processes that will communicate via sockets. Still, the sockets will be connected to each other within your computer (that is, datagrams leaving your computer will never actually leave your computer). The IP address for the loopback interface (to which the client connects) is 127.0.0.1.

- You’ll want to test your UDP client connecting to your UDP server via the loopback interface and then test your TCP client connecting to your TCP server via the loopback interface. You should test the case that (i) the connection establishment works OK, as described above; (ii) a connection request fails because a connection ID is already in use (in this case, you could run two instances of your client to force this behavior).

- Think about the answers to the following questions, but there is no need to submit your answers.
  o For your test, try two consecutive successful connections. Are client IP addresses the same? What about the client port number? Why?
  o How about the server IP address and port number? Are they the same for each connection? Why?

Part 1: What to hand in. Submit two files (one zip file and one pdf file) via GradeScope:

1. A zip file (named PA1.zip) containing the four following files:
   - Your UDP client, (please name it exactly UDP_client with its extension [e.g. .py])
   - Your UDP server, (please name it exactly UDP_server with its extension)
   - Your TCP client, (please name it exactly TCP_client with its extension)
   - Your TCP server, (please name it exactly TCP_server with its extension)

2. A separate document (in PDF format) that contains the following (along with the text explaining what the screenshots show)
- **Scenario 1:** A screenshot of a single interaction of your UDP client and server working correctly together in loopback mode. The interaction should consist of you starting up your UDP server, then starting up your UDP client, and recording whatever the client prints out.

- **Scenario 2:** Assuming neither the UDP client nor UDP server are running, start up your client alone and don’t start up the server. It should eventually timeout and display a “Connection Failure” message. Hand in a screenshot of this interaction.

- **Scenario 3:** Start up your UDP server, and then start up the first instance of your UDP client; they should interact fine, and your UDP client will exit. Now (within the next 30 seconds), start up a second instance of your UDP client that contacts the same UDP server at the same port number using the same connection ID. It should receive an error message from your server and exit with the “Connection Error <connection ID>” message. Hand in a screenshot of this interaction.

- **Scenario 4:** Lastly, start up your UDP server, and then start up the first instance of your UDP client; they should interact fine, and your UDP client will exit. Now wait for more than a minute (long enough for your server to timeout the connection ID), and then start up a second instance of your UDP client that contacts the same UDP server at the same port number using the same connection ID. They should interact fine, and your UDP client will exit. Hand in a screenshot of this interaction.

- Repeat the four steps above for your TCP client and server.

- That makes 8 screenshots in total that you’ll include in this part of the document that you will hand in. Again, please include some text on the document indicating which of these cases is being shown. The scenarios that you are recording above show that your client and server (both UDP and TCP) are interacting as required, including error and exception cases.

- Two screenshots from Part 2 (see below)

### Part 2

Once your four programs (a client/server in UDP and a client/server in TCP) are working fine in loopback mode, you can perform the second part.

In this second part, you should have your UDP and TCP clients interact with an external server that was not written by you, and that is using the IP address and port number given below. In real-world scenarios, you might be responsible for implementing either the client or the server, not both, and be responsible for interacting with code that someone else has written. But since both implementers are implementing the client and server from the same specification, they should interact, no problem! Thus, if you’ve completed Part 1 correctly, Part 2 is a piece of cake, since no new programming required!
Do the following:

- **Scenario 5 (UDP):** You should now test your UDP client with our UDP server that will be listening for your client at IP address 128.119.245.12 (this is the server gaia.cs.umass.edu), port 7500. Our UDP server will be implementing the protocol specified above. Note that if you did part 1 successfully, you just need to replace the IP address and port number with new IP/Port when you run your client. You only need to run your UDP client once (correctly interacting our server).

- **Scenario 5 (TCP):** Likewise, you should now test your TCP client with our TCP server that will be listening for clients at IP address 128.119.245.12 (this is the server gaia.cs.umass.edu), port 7500. Our TCP server will be implementing the protocol specified above. You only need to run your TCP client once (correctly interacting with our server).

- You should think about the answers to the following questions. However, there is **no need** to submit the answers to these questions.
  - Run your TCP test for two successful connections. Answer the following questions:
    - Question 2.1: What are the client IP addresses returned by the server in each of your two runs? Are they the same?
    - Question 2.2: What are the port numbers in each case? Are they the same?
  - Run your UDP test and answer the following question:
    - Question 2.3: What were the UDP client IP address and port number when you ran your client against our server? Are they the same as in your TCP runs?

**Part 2: What to hand in**

Take screenshots of your UDP and TCP client-side in action (i.e., being run, and printing out the results received from our server). Put these screenshots at the end of the same file that contains your screenshots from Part 1. Also, include text that explains what these two screenshots are doing.

- Put all these screenshots from Part 1 and Part 2 together in a pdf file with proper description for each screenshot and upload the file to GradeScope. Please remember to tag each scenario to the questions correctly on GradeScope.
- Zip 4 files of Part 1 (UDP/TCP client and server) and name the zip file as PA1.zip. Upload the zip file to GradeScope. Due to using auto-grading, make sure that filenames are exactly the same as what is mentioned in Part 1.
Programming Language

A note on programming languages:
- You are encouraged to write your client and server in Python 3 since that’s the language used in our textbook for sockets. But you can also write it in Java or C if you prefer.

Grading rubric

We’ll run the python code that you submit (both client and server) and connect them to each other. We’ll also look at the screenshots that you submit in the written document.

20  UDP client/server functioning correctly in scenario 1
6   UDP client/server functioning correctly in scenario 2
6   UDP client/server functioning correctly in scenario 3
6   UDP client/server functioning correctly in scenario 4
6   Your UDP client functioning correction in scenario 5
6   Reasonable commenting in code - identifying major sections of code.

20  TCP client/server functioning correctly in scenario 1
6   TCP client/server functioning correctly in scenario 2
6   TCP client/server functioning correctly in scenario 3
6   TCP client/server functioning correctly in scenario 4
6   Your TCP client functioning correction in scenario 5
6   Reasonable commenting in code - identifying major sections of code.