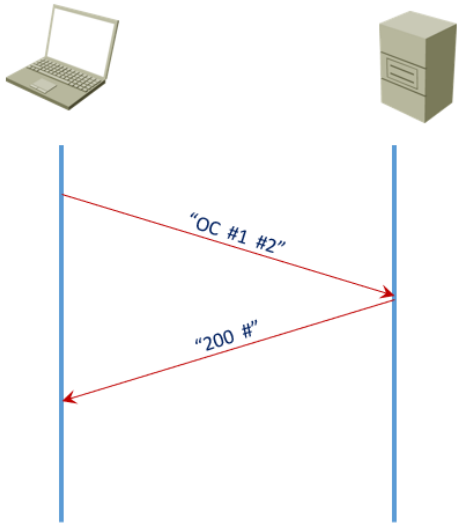
CMPSCI 453: Computer Networking

Programming Assignment 1

Version 3.0

In this assignment you will write a TCP and UDP client server program in which the server acts as a (simple) calculator to act on two numbers sent to it by the client.



Content

Part I: Simple Client Server in a reliable environment, using TCP & UDP

Write this part using both UDP and TCP (i.e., a client/server in UDP and a client/server in TCP).

The Server

The server performs the Operation Code (OC) requested on the two **integer numbers** it receives from the sender and returns the result. The format of the returned result is "status-code numeric-result" (without quote), as explained below. More specifically the steps (or algorithm) performed by the server are:

Step 1: Open a socket as a server and
Step 2: Listen to the socket.
Step 3: Receive a request which consist of an operation code (OC) and two integer numbers
Step 4: Check the OC and the numbers to make sure they are valid (OC can be one of +,-,*, and /, both numbers should be integer numbers).
Step 5: If the request is not valid, send the following error status codes with a result of -1 (to be consistent) and go to Step 2.

Invalid requests are:

- Invalid OC (i.e., not +, -, *, or /) with an error status code of 620.
- Invalid operands with an error status code of 630.
    - Operands not integer numbers (positive and/or negative)
    - Division by zero (0).

Step 6: If the request is valid, perform the operation and return a status code of 200 and the result.
Step 7: Print out a line to standard output with the following format

{received_line} -> {status_code} {result}

Step 8: Go to Step 2.
Step 9: The server should respond to ^C (Control-C) to stop.

The Client

The client sends an Operation Code (OC), and the two numbers it has acquired from the user. OC can be: Addition (+), Subtraction (-), Multiplication (*), and Division (/)

To make the problem simple, your client sends two numbers that are **integers**.

The two numbers and the OC are read from a file which the user has prepared. The algorithm is specified as follows:

Step 1: Start the client and read the name of a file as a parameter passed to it on the command line.
Step 2: From the file read a line.

- The line should contain 3 values, an Operation Code (OC) and two numbers, separated by one (or more) spaces.
- The client does not decide on the validity of the input. This is the task of the server to do.
- Display a line "Input request: { the complete input line just read}"

Step 3: Open a socket (TCP or UDP; you need to program both TCP and UDP) to the server.
Step 4: Send the line just read to the server, and wait for the results back.
Step 5: Receive the status code and the result from the server.

Step 6: If the status code is OK (i.e., 200), display the result as "The result is: {result}".
And if the status code is anything else (say 620), notify the user of the failure by displaying "Error {status_code}: {description}".
Step 7: Close the socket.
Step 8: Repeat steps 2-6 6 more times.
- All together the program reads 7 lines, each contacting 3 values separated by one or more spaces, (OC N1 N2).

Step 9: Stop the program.

To be thorough, you will handle exceptions and if anything goes wrong, notify the user and halt the program.

Please write 2 sets of programs, one using TCP (client and server), and one using UDP (client and server).

Notice that because the underlying system (your computer in fact!) is reliable, there is no need to make provision for error checking and recovery in the case of UDP. This is what you will do in the next part.

Part II: Simple Client Server in an unreliable environment, using UDP

In this part you want to simulate errors and how to handle them. When you use TCP, the transport protocol takes care of network unreliability, being dropped packets and/or erroneous transmission. Hence, we will consider only using UDP as the transport service.

But when you use UDP, your application should take care of reliability issues. **In this exercise you use your application and write the recovery code**.

Because you run both the client and the server programs on your computer, it is difficult to simulate transmission errors. Hence you simulate network/transmission error by randomly dropping (ignoring) packets at the server. The scenario is basically the same as part I, but you simulate dropping packets at the server. **To ensure reproducibility for the autograder, please use the built-in `random` module and seed it as described below.**

The UDP Server

The server performs the operation (OC) requested on the two numbers it receives from the sender and returns the result. More specifically the server must follow the following algorithm:

Step 1: Upon start, the server reads two (2) parameters from the command line: the probability *p* of dropping a received datagram, and a string for seeding a random number generator.
Step 2: Seed the random number generator by calling `random.seed(sys.argv[2])`.
Step 3: Open a socket as a server.
Step 4: Listen to the socket.
Step 5: Receive a request.
Step 6: With probability *p*: drop the request, print a line with the format "{received_line} -> dropped", and go to Step 4.
Note: To ensure reproducibility, use the expression `if random.random() <= p` to determine whether to drop a request.
Step 7: Parse the request which consist of an operation code (OC) and two integer numbers.
Step 8: Check the OC and the numbers to make sure they are valid (OC can be one of +,-,*, and /, the numbers should be both integer numbers).

Step 9: If the request is not valid, send the following error status codes and the result of -1 (to be consistent) and goes to Step 3.

Invalid requests are:

- Invalid OC (i.e. not +, -, *, or /) with an error status code of 620
- Invalid operands with an error status code of 630
    - Operands not integer numbers (positive and/or negative)
    - Division by zero (0).

Step 10: If the request is valid, perform the operation and return the result and a status code 200 and the result.

Step 11: Print out a line to standard output with the format "{received_line} -> {status_code} {result}"

Step 12: Go to Step 4

Step 13: The server should respond to ^C (Control-C) to stop.

The UDP client:

The client sends to the server an Operation Code (OC), and two numbers. OC can be: Addition (+), Subtraction (-), Multiplication (*), and Division (/). To make the problem simple, your client sends two integer numbers.

There is a possibility that the client does not receive a reply, in which case it repeats sending the request. The client uses a technique known as exponential back off, where its attempts become less and less frequent. This technique is used in other well-known communication protocols, such as CSMA/CD, the underlying protocol of Ethernet (Yes, you have been using and enjoying this protocol without knowing it!).

The client sends its initial request and waits for certain amount of time (in our case d=0.1 second). If it does not receive a reply within d seconds, it retransmits the request, but this time waits twice the previous amount, 2*d. It repeats this process, each time waiting for a time equal to twice the length of the previous cycle. This process is repeated until the wait time exceed 2 seconds. At which time, the client sends a warning that the server is "DEAD" and aborts for this request.

The two numbers and the OC are read from the user via a file. In your program

Step 1: Start the client and read the name of a file as a parameter passed to it.

Step 2: Do the following:

1. Initialize the timer value
    - Set d=0.1
2. From the file read a line.
    - The line should contain 3 values, an Operation Code (OC) and two integer numbers, separated by one (or more) spaces.
    - The client does not decide on the validity of the input. This is the task of the server to do.

Step 3: Open a UDP socket to the server.

Step 4: Send the line just read to the server.

Step 5: start a timer for d seconds, and wait for a reply to come back.

Step 6: If the timer expires (before a reply comes back),

- Set d=2*d
- If d>2,

1. Raise an exception;
2. notify the user that the server is in trouble by displaying "Request timed out: the server is DEAD", and
3. set status code to 300 (the server is dead!)
4. Go to Step 8

- Otherwise, print "Request timed out: resending" and go to Step 4

Step 7: (A reply is received before timeout.) Receive the status code and the result from the server.
Step 8: Turn off the timer.
Step 9: If the status code is 200, display the result as "Result is {result}". And if the status code is anything else, warn the user of the failure by printing "Error {status_code}: {description}".
Step 10: Close the socket.
Step 11: Repeat steps 2-10 six (6) more times.

- All together the program reads 7 lines, each contacting 3 values, (OC, N1, N2)

Step 12: Stop the program.

General

Your program should be original and not a copy. Please note that we plan to use an automated program checker for this purpose.

The input file which your program reads has 7 lines and each line contains a triplet: OC Num1 Num2 (example: + 10 12).

For each line of input, your program prints out the output in the format described above. If there is an error (e.g., a non-integer input, or a bad OC), your program should display it using the format described above. Once all input lines (7 of them) are processed, your program should stop gracefully. Note that the result of an operation (division) could be a float number which your program displays.

You are encouraged to program each process (client and server) to print out informative messages along the way for debugging, but you should disable them before submitting the code, as our autograder expects one line of output for each line of input, except when the unreliable UDP client resends a message.

Write your code generically, e.g., use 127.0.0.1 as your server address (assuming your client and server are on the same computer) so your code can be used on any computer.

Your server processes must catch interrupts of Ctrl-C and exit gracefully without printing a backtrace. Example code for this is below:

```
try:
   while True:
      # Perform socket operations
except KeyboardInterrupt:
   # Close sockets
```

We will test your program with a file of 7 input lines; each line containing (OC Num1 Num2). All programs are tested similarly.

Please write your program in a supported version of Python 3. You can use any environment of your choice, Unix, Linux, Mac OS, or Windows.

A good reference for Python socket programming is
http://docs.python.org/howto/sockets.html

Another good book for Python network programming is
"*Foundation of Python Network Programming*", 3rd Ed. By Brandon Rhodes and John Goerzen, Apress, 2014. I have a copy of this book. You are welcome to use/read this book in my office (sorry, I cannot lend it out; this is my only copy and I refer to it frequently; my apologies).

You can use any reference material that you like, work together (which I encourage), but submissions should be done individually. PLEASE make sure your submission is yours. I count on everyone's honesty. Use this as an opportunity to learn.

Rubric

All together you will submit 6 programs

a. TCP Client/Server (2 Programs)
b. Reliable UDP Client/Server (2 programs)
c. Unreliable UDP Client/Server (2 Programs)

Each pair of programs is tested against 7 set of values. All submissions are tested against the same set, in the same order.

Your submission is graded as follows:

Program	Program does not crash	Correct Result for a (OC N1 N2)	Total Credit
TCP Client/Server	9%	3%	9%+(3% * 7) = 30%
Reliable UDP Client/Server	9%	3%	9%+(3% * 7) = 30%
Unreliable UDP Client/Server	19%	3%	19%+(3% * 7) = 40%

If a program crashes during a test, it is entitled for credit only for the tests which ran successfully.

Your Submissions, Naming Conventions, Tips & Sample Data

Files You Submit

Altogether, you write 6 programs:

- TCP_Client.py, TCP_Server.py (2 programs)
- UDP_Client-Reliable.py, UDP_Server-Reliable.py (2 programs)
- UDP_Client-Unreliable.py, UDP_Server-Unreliable.py (2 programs)

Please put ALL programs in **SINGLE** .zip file and post it on Gradescope.

Naming Conventions

Please name you zip file as following

<last-name>_<first_name>.Client-Server-Claculator.zip

For example, my submission would be names

Kermani_Parviz.Client-Server-Calculator.zp

Please name your 6 programs as:

- TCP_Client.py, TCP_Server.py
- UDP_ Reliable-Client.py, UDP_Reliable-Server.py
- UDP_Unreliable-Client.py, UDP_Unreliable-Server.py

Tips

- You must choose a server port number greater than 1023 (to be safe, choose a server port number larger than 50000).
- I would strongly suggest that everyone begin by writing one client and one server first, i.e., just getting the two of them to interoperate correctly.
- To specify your server's identity, you can
  a. Use the name "localhost", which Is translated to your own computer.
  b. Use your computer's generic IP address, 127.0.0.1
  c. Use your own computer's IP address. You can find your own IP address by using commands like `ipconfig /all` (on Windows), `ipconfig` (on Linux, Unix, macOS) or `ip address show` (on newer Linux, like Ubuntu 22.04).

  You are strongly advised not to use option (c), because it binds your program to the specific IP address. If you change your location, you will most probably have another IP address and your program does not work.

  Option (b) is the most preferred one. You can try running your client and server programs on 2 separate computers or VMs.

- Many of you will be running the clients and senders on the same UNIX/Linux machine (e.g., by starting up the receiver/server and running it in the background) then starting up the sender/client. This is fine; since you are using sockets for communication, these processes can run on the same machine or different machines without modification. Recall the use of the ampersand (&) to start a process in the background (in Linux and Unix). If you need to kill a process after you have started it, you can use the UNIX kill command. Use the UNIX `ps` command to find the process id of your server.
- Make sure you close every socket that you use in your program. If you abort your program, the socket may still hang around and the next time you try and bind a new socket to a port which you previously used (but never closed), you may get an error. Also, please be aware that port numbers, when bound to sockets, are system-wide values and thus other programs may be using the port number you are trying to use.

- I strongly suggest that once your program is running, use 2 computers and you can use Wireshark to see the flow, it is fun!

Sample Data

We will test ALL programs with the same input date. Each pair of programs is tested against the same 7 set of values.

The input value in stored in a text file. Your program will read the name of the data file as a parameter.

Here we show a sample date. Please put it in a text file and run your programs against it to make sure we can test them:

```
+ 2    10
- 100  20
* 25  -3
% 200  3
- 2.5  15
/ 65   4
+ 0    12
```

Good luck to you all …. PK