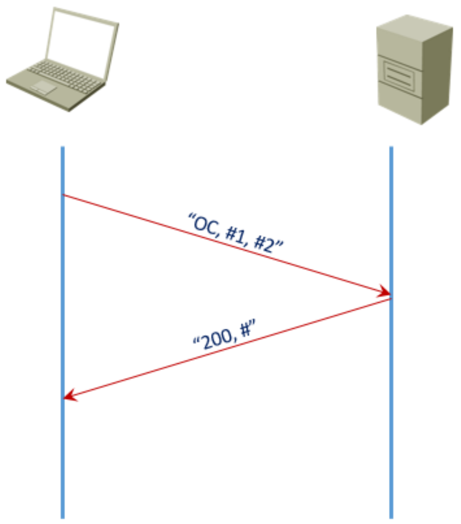
CMPSCI 453: Computer Networking
Fall 2019
Programming Assignment 1

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.



Part I: Simple Client Server in a reliable/unreliable environment

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. More specifically the steps (or algorithm) performed by the server are:

1. Open a socket as a server and
2. Listen to the socket
3. Receive a request which consist of an operation code (OC) and two integer numbers
4. 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).
5. If the request is not valid, send a return status code of "300" (or any number which you decide upon) and the result of -1 (to be consistent) and goes to Step 2.
   Invalid requests are:
   - Invalid OC (i.e. not +, -, *, or /)
   - Invalid operands

    - Operands not integer numbers (positive and/or negative)
    - Division by zero (0).

6. If the request is valid, perform the operation and return the result and a status code of (e.g. 200 or any number which you decide upon) and the result.
7. Go to Step 2
8. 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:

1. Start the client and read the name of a file as a parameters passed to it on the command line.
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.
3. Open a socket (TCP or UDP; you need to program both TCP and UDP) to the server.
4. Send the line just read to the server, and wait for the results back
5. Receive the status code and the result from the server.
6. If the status code is OK (e.g. 200), display the result. And if the status code is anything else (say 300), notify the user of the failure.
7. Close the socket
8. Repeat steps 2-6 five more times
    - All together the program reads 6 lines, each contacting 3 values, (OC, N1, N2).
9. Stop the program.

To be thorough, you will handle exceptions and if anything goes wrong, notify the user and halt the program. You can also choose any code to represent the OC, the Operation Code. This becomes part of your “Application Layer Protocol”.

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

Because you run both client and server on your computer, it is difficult to simulate transmission error. 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.

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:

1. Upon start, the server reads one (1) parameter from the command line which is the probability *p* of dropping a received datagram.
2. Open a socket as a server and
3. listen to the socket
4. Receive a request
5. With probability *p* drop the request and go to step 3.
6. Parse the request which consist of an operation code (OC) and two integer numbers
7. 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).
8. If the request is not valid, send a return status code of “300” (or any number which you decide upon) and the result of -1 (to be consistent) and goes to Step 3.
   Invalid requests are:
   - Invalid OC (i.e. not +, -, *, or /)
   - Invalid operands
     - Operands not integer numbers (positive and/or negative)
     - Division by zero (0).
9. If the request is valid, perform the operation and return the result and a status code of (e.g. 200 or any number which you decide upon) and the result.
10. Go to Step 3
11. The server should respond to ^C (Control-C) to stop.

The UDP client:

The client sends 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 and 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

1. Start the client and read the name of a file as a parameters passed to it.
    - Set d=0.1
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
2. Open a UDP socket to the server.
3. Set *d*=0.1,
3. Send the line just read line to the server, and go to the next step
4. start a timer for *d* seconds, and wait for a reply to come back
5. If timeout expires (before a reply comes back),
    - Set *d*=2*d
    - If d>2,
        - Raise an exception;
        - notify the user that the server is in trouble, and
        - Go to Step 7
    - Otherwise go to step 3
6. (A reply is received before timeout.) Receive the status code and the result from the server
7. Turn off the timer.
8. If the status code is OK (say 200), display the result. And if the status code is anything else (say 300), warn the user of the failure.
9. Close the socket
10. Repeat steps 2-6 five more times
    - All together the program reads 6 lines, each contacting 3 values, (OC, N1, N2)
11. 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 6 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 a readable format. E.g. “The result of 10 + 12 is: ….. “. If there is an error (e.g. a non-integer input, or a bad OC), your program should display it clearly and move on to the next input line. Once all input line (6 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 should program each process (client and server) to print out informative messages along the way. This helps you to follow your program and debug it and helps me and the grader to verify that your program is working fine.

Write your code generically, i.e. 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.

We will test your program with a file of 6 input lines; each line containing (OC, Num1, Num2). All programs are tested similarly.

Please write your program in Python (preferably 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 submission 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 program is tested against 5 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	6%	4%	6%+(4% * 6) = 30%
Reliable Client/Server	6%	4%	6%+(4% * 6) = 30%
Unreliable Client/Server	16%	4%	16%+(4% * 6) = 40%

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

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.

- You may need to know your machine's IP address, when one process connects to another. You can use “`ipconfig /all`” command on Windows or “`ifconfig`” command on Linux/Unix/Mac OS. Alternatively, you can use the name of your server. But remember, the name should be resolvable to an IP address. Should you decide to use a name, please use the name “`localhost`” so we can test your program on any computer.
- 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 & 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 the port 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 so that you can use Wireshark to see the flow, it is fun!

Good luck to you all …. PK