ICSI516 Spring 2017 -- Project 1
Due: Monday, March 6th by 11:59pm via Blackboard

Objectives

There are a number of objectives to this assignment. The first is to make sure you have some experience developing a network-based socket application. Second, because you are allowed to use any references you find on the Internet (including copies of existing code!), this assignment will help you see just how many network programming aids are available. Third, you will get a first-hand experience in comparative evaluation of protocol performance. Finally, having just a bit of practical experience will put a lot of the protocol concepts we learn into perspective.

Reading

I would recommend you go over Chapter 2.7 before you begin your implementation. You can also use various online resources on socket programming.

Assignment Overview

This assignment asks you to develop and evaluate a web-based calculator for simple expressions. You will complete this assignment in two phases. In the first phase you will develop two versions of the calculator: one that uses TCP as a transport protocol, and one that uses UDP for transport and implements basic stop-and-wait reliability at the application layer. In the second phase you will evaluate and compare the performance of the two implementations in terms of achieved throughput and delay.

Assignment Details

============== Phase 1 -- Implementation ==============

Note: All your code should compile and run on itsunix. Programs that have been developed on your personal computer and do not run on itsunix will not be graded and will receive 0 points.

The goal of this assignment is to implement a simple client-server system. You can use Python or Java. The basic functionality of the system is a remote calculator. You first will be sending expressions through the client to the server. The server will then calculate the result of the expression and send it back to the client where it is displayed. The server should also send back to the client the string **"Socket Programming"** as many times as the absolute value of the result from the expression.

Your server will start in a "passive mode", in other words, it will listen to a specified port for instructions from the client. Separately, the client will be started and will connect to the server on a given host name or IP address plus a port number. The client should take as input from the user the host name or IP address plus the port number and an expression to be calculated. Consider solving an expression that would generate relatively big result, allowing you to test transmission of multiple packets (see TCP and Reliable UDP below for more details).

The client, when started, should request the following information:

1. Enter server name or IP address:
2. Enter port:
3. Enter expression:

and should accept a single line for each query.

The client should make sure that the specified port is valid, in other words, the client should check if the port is in the range from 0 to 65535 and if not should display a message of the format **"Invalid port number. Terminating."**

The client, then, will try to connect to the specified server on the specified port and transmit the expression to the server.

The interaction flow between the client and server is as follows:

1. The server starts and waits for a connection to be established by the client.
2. When an expression is received, the server will:
   - Find the result of the expression and store it in a local variable.
   - Write the result and the corresponding number of strings **"Socket Programming"** to the connection established by the client.
3. Finally, the client will receive the result from the socket and display it to the user.

Your remote calculator should support the four basic mathematical operations plus various combinations of them:

1. Addition:
   - 3 + 2
2. Subtraction:
   - 3 - 2
3. Multiplication:
   - 3 * 2
4. Division:
   - 3 / 2
5. Combination:
   - (3 + 7) / 2

Note that you have to handle network errors. Make sure that you verify in your client code if the connection to the server was successful. If the connection failed, your client program should display an error of the format: "**Could not connect to server. Terminating.**"

Also make sure your client returns an error if the result was not successfully received from the server. The client should display the following text: "**Could not fetch result. Terminating.**"

TCP and Reliable UDP

You will implement your client and server using both TCP and UDP. This means you will be writing four different programs. (The clients and servers using TCP and UDP won't be interoperable so you won't be able to use the TCP client with the UDP server and vice versa. As food for thought and discussion in the future, why do you think these different protocols won't interoperate?) Using TCP should be more straightforward since it has connections, reliability, in-order delivery, etc. However, UDP by itself does not include any of these features so you will have to add a simple form of reliability. You will do this by applying the stop-and-wait principle. Since we are working at the application level and there is no concept of packets, we will implement our stop and wait reliability at the level of character strings.

The client and server using Reliable UDP will operate as follows:

1. Your client will format the expression to be calculated and send the length of the expression to the server (a "length" message). Then, in a separate transmission, the client will send the expression string.

2. When the server receives the length message it will wait up to 500 milliseconds for that number of bytes to be sent. If it receives the correct number of bytes, it will respond with a string containing the characters "ACK" (a common abbreviation for an

acknowledgement). If it does not receive the correct number of bytes by the end of the timeout period, the server will give up, display the following text on the server console, and exit: "**Did not receive valid expression from client. Terminating.**" If you have a large result that is broken up into segments, every segment must be ACKed in order to preserve the stop-and-wait characteristics of the protocol.

3. If your client has not received an ACK within 1 second it will resend the length value and then the expression again.

4. Your client will send, up to 3 times, before giving up, closing the connection and terminating. Upon closing connection, the client should display the following message: "**Failed to send expression. Terminating.**"

5. The server will implement reliability in a similar fashion. First, it will compute the size of the result it has to send and then will send it to the client in a "length" message. The server will then send messages containing the output one by one and will wait for up to 1 second to receive an ACK for each message sent, before sending the next one. Similar to the client, the server will retry sending a single message up to three times before giving up. If the server did not receive an ACK after three attempts, it should display the following text on the server console and exit: "**Result transmission failed. Terminating.**"

6. Finally, upon successful reception of the output, the client should display the result.

   NOTE: When sending large amounts of data over UDP, you might need to transmit multiple portions. This is because the UDP buffer fills up before the total amount of data is sent. Let's look at an example - say that your server needs to send a total of 5120 bytes but the UDP buffer is only 512 bytes. In this case, even though your server said that it is sending 5120 bytes, the UDP on the client delivers the data in 512 byte chunks because of the buffer size. A program that sends an ACK only if the received number of bytes is the same as the expected number of bytes will fail to send an ACK in this case, which eventually will cause the connection to timeout and the server will fail to send result to the client. To avoid such problems, your programs will have to send an ACK for each received portion of data, keep track of how much of the entire message had been received and display the result only if the total amount received is equal to the value sent in the length message. This problem will probably not occur when your client sends data to your server because the entire expression will fit into a single packet and thus it will arrive all at once. However, you will want to test your system with large enough results to confirm that it works correctly.

Examples

Below are some output examples, to help you format your programs display. The format of both the TCP and UDP programs should be the same - that is why we don't have separate examples for TCP and UDP. The only difference between running TCP and UDP programs is going to be the names of the programs to be run (see section "Language Choice and File Names" for naming conventions). Because we only have access to one development machine (itsunix), you will be running both the server and the client on the same physical machine. For this purpose you need to use the localhost address (or 127.0.0.1) when running your programs.

Starting the Server

You are working on itsunix. You should start the server by issuing a command in a terminal, whose syntax should be as follows (see below for correct naming of executables):

```
unix1> server 3300
```

where "server" is the executable for the TCP or UDP program you wrote and "3300" is the port number you are going to use to establish connection.

Client Input/Output Example

You are working on itsunix. You should start the client by issuing the following command in a terminal (see below for correct naming of executables):

```
unix1> client
```

where client is the executable for the TCP or UDP program you wrote.

Here are some other examples of what the client might display depending on the user input.

```
unix1> client
Enter server name or IP address: gizmo.cs.albany.
Could not connect to server. Terminating.
unix1>
```

or

```
unix1> client
Enter server name or IP address: 428.163.7.19
Could not connect to server. Terminating.
unix1>
```

or

```
unix1> client
Enter server name or IP address: 128.163.7.19
Enter port: 99999
Invalid port number. Terminating.
unix1>
```

or

```
unix1> client
Enter server name or IP address: 127.0.0.1
Enter port: 3300
Enter expression: 2 + 3

Failed to send expression. Terminating.
unix1>
```

or

```
unix1> client
Enter server name or IP address: 128.163.7.19
Enter port: 3300
Enter expression: 2 + 3

Could not fetch result. Terminating.
unix1>
```

or

```
unix1> client
Enter server name or IP address: localhost
Enter port: 3300
Enter expression: 2 + 3

5
Socket Programming
Socket Programming
Socket Programming
Socket Programming
Socket Programming
unix1>
```

Server Output Examples

```
unix1>server 3300
Did not receive valid expression from client. Terminating.
unix1>
```

or

```
unix1>server 3300
Result transmission failed. Terminating.
unix1>
```

or

```
unix1>server 3300
unix1>
```

(In the case of correct operation, no output is displayed on the server console.)

Language Choice and File Names

You will choose Python or Java for this assignment. Regardless of the language choice, you must turn in exactly four programs (all headers, etc. should be included in one file). **Your programs should be named as indicated below and should compile with NO errors and warning and run on itsunix!**

For Java, the program names should be:

- Client in Java using TCP: client_java_tcp.java
- Server in Java using TCP: server_java_tcp.java
- Client in Java using UDP: client_java_udp.java
- Server in Java using UDP: server_java_udp.java

To compile your Java code, use the following commands:

- "*javac client_java_tcp.java*"
- "*javac server_java_tcp.java*"
- "*javac client_java_udp.java*"
- "*javac server_java_udp.java*"

For Python, the program names should be:

- Client in Python using TCP: client_python_tcp.py
- Server in Python using TCP: server_python_tcp.py
- Client in Python using UDP: client_python_udp.py
- Server in Python using UDP: server_python_udp.py

To compile your Python code, use the following commands:

- "*python client_python_tcp.py*"
- "*python server_python_tcp.py*"
- "*python client_python_udp.py*"
- "*python server_python_udp.py*"

NOTE: Pay attention to all of these directions carefully. Make sure that you name your files and format your messages as specified in the assignment. An automated program will be used for grading and if there are any deviations, you will lose points!

============== Phase 1 -- Wireshark evaluation ==============

In this phase you will perform a comparative evaluation of your implementations in terms of overall delay and achieved throughput. We define overall delay as the relative time difference between the last and the first packet exchanged within a single program invocation. We define the achieved throughput as the total sum of bits exchanged within a single program invocation divided by the overall delay for that invocation. You will run your server and client implementations on different physical machines in order to account for a realistic Internet scenario. Specifically, you will run your server program on itsunix and your client program on your personal computer. You will also run Wireshark on your personal computer to be able to record a packet trace for each program invocation. You will need to record four packet traces for each of the TCP and UDP implementation (so eight altogether), as specified in the tables below. Once you have the traces, you need to process them offline and determine the overall delay and achieved throughput for each invocation.

Expression/Implementation	1+1	5+5	20+20	50+50
Overall delay (TCP), s				
Overall delay (UDP), s				

Expression/Implementation	1+1	5+5	20+20	50+50
Achieved throughput (TCP), bps				
Achieved throughput (UDP), bps				

For this phase you need to submit a brief report (not more than 2 pages) on your findings. This report should include the following sections:

- *Two tables, using the same format as the ones above, with filled out values for overall delay and achieved throughput, calculated in your Wireshark analysis.*
- *A description of the trends you see in your results along with a justification of these trends.*

Note: Consider running your experiments from the same network. For example, if you run your UDP experiments from campus and your TCP experiments from home, different delay characteristics of the campus and your home network will skew your results.

Grading Guidelines

You may use pieces of code from the Internet to help you do this assignment (e.g. basic socket code). However, this is just like citing a passage from a book, so if you copy code, you **must** cite it. To do this, put a comment at the beginning of your code that explains exactly what you have copied, who originally wrote it, and where it came from.

Below is a breakdown of points for this assignment. In addition to correctness, part of the points count towards how well code is written and documented. NOTE: good code/documentation does not imply that more is better. The goal is to be efficient, elegant and succinct!

- 25 pts: Clients (TCP and Reliable UDP)
- 35 pts: Servers (TCP and Reliable UDP)
- 10 pts: Documentation/Proper References
- 30 pts: Wireshark evaluation

Final Warning

This is an assignment you definitely want to start on early. The design of the assignment is such that it is nearly impossible to provide all of the details you need. Instead of assuming things should be done a particular way, **ask questions**! Use every opportunity to meet with the instructor and TAs and send them emails with questions. Answers that are relevant to everyone in class will be posted on the Blackboard discussion forum.

Assignment Turnin

The assignment should be submitted using the course Blackboard page. You need to submit a total of 13 files as follows:

- The programs for your UDP and TCP implementation (4 files).
- A 2-page report on your evaluation (1 file).
- Your pcap traces (8 files).

Because the web site only allows one file to be submitted, you should use zip to combine all your files into a single archive and submit this archive for grading.

Cheating Policy

Cheating is not tolerated. Please, read the university Community Rights and Responsibilities for more information on cheating. Students caught cheating will receive 0 points for the assignment and will be reported. Of particular relevance to this assignment is the need to properly cite material you have used. Failure to do so constitutes plagiarism.