Computer Networks
Fall Semester 2016

Using Socket API to Measure TCP performance

DDL :

In this assignment, you will first write a program that uses the socket interface to send messages between a pair of machines, namely, a client and a server. This first part will familiarize you with basic socket programming if you have not been exposed to it before. Then, you will use this program to measure the round trip time and throughput of a TCP connection between the client and the server. The second part of the assignment is meant to introduce you to basic network measurements.

Part I: Writing an Echo Client-Server Application

Overview

For this part, you will implement a client and a server that communicate over the network using TCP. The server is essentially an echo server, which simply echoes the message it receives from the client. Here is what the server and client must do:

> The server should accept, as a command line argument, a port number that it will run at. After being started, the server should repeatedly accept an input message from a client and send back the same message.
>
> The client should accept, as command line arguments, a hostname (or IP address), as well as a port number for the server. Using this information, it creates a connection (using TCP) with the server, which should be running already. The client program then sends a message (text string) to the server using the connection. When it receives back the message, it prints it and exits.

In Part I, the message is simply a text string with no specific format. In other words, any text message will do. You may use C, C++, or Java to build your client and server programs. In C/C++, you should familiarize yourselves with the following system calls: *socket(), select(), bind(), listen(), accept(), connect()*, *send()* and *recv()*. We outline a number of resources below with additional information on these system calls, as well as their equivalent in several other programming languages.

What to submit

The programs you submit should work correctly and be well documented with a record of the information exchanged between your client and server. It would be a good idea to test your client program with the server program implemented by a classmate, or vice versa. This is possible because your programs should adhere to the same exchange protocol described above. This is also an interesting way to see how protocol modules can "interoperate" if they accurately implement the same protocol specification. *You must, however, implement both the client and server on your own*.

Notes

This primer is an excellent introduction to BSD sockets. A good book is also available on this topic.

If you are using Java, you may look at the UDP Pinger Lab at the end of Chapter 2 of the K&R textbook. Code for the UDP ping/echo server is provided and may be helpful.

A short tutorial on socket programming, from the University of Wisconsin:
ftp://gaia.cs.umass.edu/cs653/sock.ps

Additional socket programming links:
http://compnetworking.about.com/cs/socketprogramming/
Network Programming with Sockets

Part II: Performing RTT and Throughput Measurements

Overview

In this part, you will extend the echo application implemented in Part I to measure the round trip time (RTT) and throughput of the path connecting the client to the server. To measure RTT, you will use TCP to send and receive messages of size 1, 100, 200, 400, 800 and 1000 bytes. To measure throughput, you will use TCP to send and receive messages of size 1K, 2K, 4K, 8K, 16K and 32K bytes. Note the difference in units for the two sets of message sizes. For each measurement and for each message size, the client will send at least ten probe messages to the server which will echo back the messages.

Protocol Phases

As in Part I, the client first needs to set up a TCP connection to the server using the socket interface. The echo application will be extended, however, by specifying the exact protocol interactions between the client and the server. This entails specifying the exact message formats, as well as the different communication phases, as outlined next.

1) *Connection Setup Phase (CSP)*

This is the first phase in the protocol where the client informs the server that it wants to conduct active network measurements in order to compute the RTT and throughput of its path to the server. We will outline the expected behavior from both the client and the server next.

CSP: Client

After setting up a TCP connection to the server, the client must send a single message to the server having the following format:

<PROTOCOL PHASE><WS><MEASUREMENT TYPE><WS><NUMBER OF PROBES><WS><MESSAGE SIZE><WS><SERVER DELAY>

PROTOCOL PHASE

The protocol phase during the initial *setup* will be denoted by the lower case character 's'. This allows the server to differentiate between the different protocol phases that the client can be operating at, as we will see.

MEASUREMENT TYPE

Allows the client to specify whether it wants to compute the RTT, denoted by "rtt" or the throughput, denoted by "tput".

NUMBER OF PROBES

Allows the client to specify the number of measurement probes that the server should expect to receive. Once all the probe messages have been echoed back and a sample measurement is taken for each one, the client should compute an estimate of the mean (average) RTT or mean throughput, depending on the type of measurement being performed. A detailed description of the probe message's format is provided in the description of the Measurement Phase.

MESSAGE SIZE

Specifies the number of bytes in the probe's payload.

SERVER DELAY

Specifies the amount of time that the server should wait before echoing the message back to the client. The default value is 0. You will vary this value later to emulate paths with longer propagation delays. Even though increasing the server delay merely increases the processing time at the server, it nevertheless causes the feedback delay, observed by the sender, to increase which has an effect somewhat similar to increasing the path's propagation delay.

WS

White space to separate the different fields in the message. The white space could serve as a delimiter for the server when parsing or tokenizing the received message.

CSP: Server

The server should parse the connection setup message to log the values of all the variables therein since they will be needed for error checking purposes. Upon the reception of a valid connection setup message, the server should respond with a text message containing the string "200 OK: Ready" informing the client that it can proceed to the next phase. On the other hand, if the connection setup message is incomplete or invalid, the server should respond with a text message containing the string "404 ERROR: Invalid Connection Setup Message" and then terminate the connection.

CSP: Summary

During correct operation, after setting up a TCP connection to the server, the client sends a single connection setup message to the server. The server parses and logs the information in the message and responds with a "200 OK: Ready" text message informing the client to proceed to the next phase.

2) *Measurement Phase (MP)*

In this phase, the client starts sending probe messages to the server in order to make the appropriate measurements required for computing the mean RTT or the mean throughput of the path connecting it to the server. We will outline the expected behavior from both the client and the server next.

MP: Client

The client should send the specified number of probe messages to the server with an increasing sequence number starting from 1. More specifically, the message format is as follows:

<PROTOCOL PHASE><WS><PROBE SEQUENCE NUMBER><WS><PAYLOAD>

PROTOCOL PHASE

The protocol phase when conducting the *measurements* will be denoted by the lower case character 'm'.

PROBE SEQUENCE NUMBER

The probe messages should have increasing sequence numbers starting from 1 up to the number of probes specified in the connection setup message using the NUMBER OF PROBES variable.

WS

White space

MP: Server

The server should echo back every probe message received. It should also keep track of the probe sequence numbers to make sure they are indeed being incremented by 1 each time and do not exceed the number of probes specified in the connection setup phase. If the probe message is incomplete or invalid (contains an incorrect sequence number for example) the server should not echo the message back. Instead, the server should respond with a text message containing the string "404 ERROR: Invalid Measurement Message" and then terminate the connection.

MP: Summary

Client repeatedly sends measurement messages to the server in an attempt to compute the mean RTT and/or mean throughput. A sample measurement is taken for each probe sent out. Server repeatedly echoes messages back to the client unless it detects erroneous behavior in which case it sends an error message and terminates the connection.

3) *Connection Termination Phase (CTP)*

In this phase, the client and the server attempt to gracefully terminate the connection. We will outline the expected behavior from both the client and the server next.

CTP: Client

The client should send a termination request to the server and then wait for a response (unless of course the server already terminated the connection due to an error in the Measurement Phase). Once a response is received, the client should terminate the connection. The message format is as follows:

<PROTOCOL PHASE><WS>

PROTOCOL PHASE

The protocol phase when *terminating* the connection will be denoted by the lower case character 't'.

WS

White space

CTP: Server

If the message format is correct, the server should respond with a text message containing the string "200 OK: Closing Connection". Otherwise, the server should respond with a text message containing the string "404 ERROR: Invalid Connection Termination Message". Either way the server should terminate the connection.

CTP: Summary

During correct operation, client sends termination message, server responds with "200 OK: Closing Connection" text message and then both terminate the connection.

Notes

Make sure to report throughput numbers (e.g., 500 kbps) and not just the amount of time it takes to exchange some number of bytes.

Give a thorough description of your experiments, including what kind of machines (e.g. the names of the CS Linux machines you used), operating systems, network, statistics collection method (including how many times the experiments are repeated before the average is taken), etc.

You may use the *gettimeofday()* system call or something similar to get the time.

Collaboration

You will have 2 weeks for the first part and 6 weeks for the second part.
Please make sure that absolutely everything what you submit is YOURS. I will not make any exception nor be lenient to a violation of any kind to plagiarism rules.

Submission

Put all the three parts in folders part1, part2 respectively. Then ZIP the folders in an archive which bears your EPOKA login name. Email that to me by the stated deadline.