CSE 589 Spring 2014
Programming Assignment 1
Remote File Sharing System

Due Time : 02/28/2014 @ 23:59:59

1. Objective

Getting Started: Familiarize yourself with socket programming.
Implement: Develop a simple application for file sharing among remote hosts and observe some network characteristics using it.
Analyze: Understand the packet-switching network behavior and compare the results of your file sharing application with those of a standard tool, iperf, used for measuring network performance.

2. Getting Started

2.1 Socket Programming

Beej Socket Guide: http://beej.us/guide/bgnet

2.2 iPerf

Iperf Homepage: http://iperf.sourceforge.net/

3. Implement

3.1 Programming environment

You will write C (or C++) code that compiles under the GCC (GNU Compiler Collection) environment. Furthermore, you should ensure that your code compiles and operates correctly on the CSE student servers (timberlake.cse.buffalo.edu, euston.cse.buffalo.edu, embankment.cse.buffalo.edu, underground.cse.buffalo.edu and highgate.cse.buffalo.edu). This means that your code should properly compile by using the version of g++ (for C++ code) or gcc (for C code) found on the CSE student servers and should function correctly when executed.

NOTE: *If you're using C++, you are not supposed to use any STL's (Standard Template Library) for the socket programming part.. If you've used any STL's for the socket programming part, then your whole project will not be graded.*

3.2 Running your program

Your process (your program when it is running in memory) will take 2 command line parameters. The first parameter indicates whether your program instance should run as a server or a client. The second parameter corresponds to the port on which your process will listen for incoming connections. (e.g., if your program is called prog1, then you can run it like this: ./prog1s 4322, where the "s" indicates that it is the server and 4322 is the port. Suppose you want to run it as a client then you should run it as ./prog1

c 4322 where the "c" parameter indicates that the process is a client and 4322 is the listening port.)

NOTE: Use TCP Sockets only for your implementation. You should **use only the select() API** for handling multiple socket connections. Please don't use multi-threading or fork-exec.

NOTE: Please run your **server version of the program only on timberlake.cse.buffalo.edu**. Your client versions should be stored/tested on any of these 4 servers only:
underground.cse.buffalo.edu
euston.cse.buffalo.edu
embankment.cse.buffalo.edu
highgate.cse.buffalo.edu

Also, create your own directory in /local/ directory to store your programs/files on the above 4 servers. **Use only this directory to store/run your programs.** However, on timberlake use your default (home) folder.

Please don't use any other folders to store/test your programs. Also, make sure you have set appropriate permissions to your folder so that it cannot be accessed by any other students.

3.3 Functionality of your program

When launched, your process should work like a UNIX shell. It should accept incoming connections and at the same time provide a user interface that will offer the following command options: (Note that specific examples are used for clarity.)

1. HELP Display information about the available user interface options.

2. MYIP Display the IP address of this process.
Note: The IP should not be your "lo" address (127.0.0.1). It should be the actual IP of the server.

3. MYPORT Display the port on which this process is listening for incoming connections.

4. REGISTER <server IP> <port_no>: This command is used by the client to register itself with the server and to get the IP and listening port numbers of all the peers currently registered with the server.

The first task of every client is to register itself with the server by sending the server a TCP message containing its listening port number. The server should maintain a list of the IP address and the listening ports of all the registered clients. Let's call this list as "Server-IP-List". Whenever a new client registers or a registered client exits, the server should update its Server-IP-List appropriately and then send this updated list to all the registered clients. Client should always listen to such updates from the server and update their own local copy of the available peers. **Any such update which is received by the client should be displayed by the client.** The REGISTER command takes 2 arguments. The first argument is the IP address of the server and the second argument is the listening port of the

server.

Note: The REGISTER command works only on the client and should not work on the server.

5. CONNECT <destination> <port no>: This command establishes a new TCP connection to the specified <destination> at the specified < port no>. The <destination> can either be an IP address or a hostname. (e.g., CONNECT euston.cse.buffalo.edu 3456 or CONNECT 192.168.45.55 3456). The specified IP address should be a valid IP address and listed in the Server-IP-List sent to the client by the server. Any attempt to connect to an invalid IP/hostname or an IP/hostname not listed by the server in its Server-IP-List should be rejected and suitable error message should be displayed. Success or failure in connections between two peers should be indicated by both the peers using suitable messages. Self-connections and duplicate connections should be flagged with suitable error messages. Every client can maintain up-to 3 connections with its peers. Any request for more than 3 connections should be rejected. However, every client should always maintain a constant connection with the server.

6. LIST Display a numbered list of all the connections this process is part of. This numbered list will include connections initiated by this process and connections initiated by other processes. The output should display the hostname, IP address and the listening port of all the peers the process is connected to. Also, this should include the server details.

E.g., id: Hostname IP address Port No.

1: timberalake.cse.buffalo.edu	192.168.21.20	4545
2: embankment.cse.buffalo.edu	192.168.21.21	5454
3: highgate.cse.buffalo.edu	192.168.21.22	5000
4: euston.cse.buffalo.edu	192.168.21.22	5000

Note: The connection id 1 should always be your server running on timberlake.cse.buffalo.edu. The remaining connections should be the peers who you are connected to.

7. TERMINATE <connection id.> This command will terminate the connection listed under the specified number when LIST is used to display all connections. E.g., TERMINATE 2. In this example, the connection with embankment should end. An error message is displayed if a valid connection does not exist as number 2. If a remote machine terminates one of your connections, you should also display a message.

8. EXIT Close all connections and terminate this process**.** When a client exits, the server de-registers the client and sends the updated "Server-IP-List" to all the clients. Clients on receiving the updated list from the server should display the updated list.

9. UPLOAD <connection id.> <file name> For example, UPLOAD 3 /local/a.txt. This will upload the file a.txt which is located in /local/ to the host on the connection that is designated by number 3 when LIST is used. An error message is displayed if the file was inaccessible or if 3 does not represent a valid connection. **The remote machine will automatically accept the file and save it under the original name in the same directory where your program is.** When the upload is complete this process will display a message indicating so. Also, the remote machine will display a message in its user interface indicating that a file (called a.txt) has been downloaded. When an upload is occurring, the user interface of the uploading process will remain unavailable until the upload is complete. Upon completion, a message is displayed. If the upload fails for some reason, an error message should be displayed. When an upload is occurring, a message should be displayed on the remote machine when the upload begins. If the upload fails for some reason, an error message should be displayed on the remote machine.

At the end of each successful upload, you will print the rate at which the Transmitter (Sender/Uploader) uploaded the file. Similarly, you will also print at the receiver, the rate at which the file was received. We will call these as Tx Rate and Rx Rate. Tx rate is the amount of data (in Bits) transferred from the Tx end, divided by the time taken by Tx to read and send the complete file in chunks of *Packet Size* bytes each. The Rx rate is defined similarly (in Bits/sec) as the total file size received (in bits) divided by the time taken to receive the file over the socket and write it. For example, if a file was uploaded from embankment to euston, the format for printing this information is:

At Tx end:

Tx (embankment): embankment -> euston, File Size: x Bytes, Time Taken: y seconds, Tx Rate: z bits/second.

At Rx end:

Rx (euston): embankment -> euston, File Size: a Bytes, Time Taken: b seconds, Rx Rate: c bits/second.

NOTE: You should read the file in chunks of *packet size-byte* buffers and send those buffers using the send socket call, instead of reading the whole file at once. You can use UNIX utility function *gettimeofday* to know the time taken for receiving and sending the file at the two ends. Make sure to appropriately call this function in your program to account only for the time taken for uploading (reading and sending) and downloading (receiving and writing) the file. After printing this information, flush the standard output using *fflush*, to immediately print this information.

NOTE: UPLOAD command on a server should display an error message. No files should be uploaded to the server. Any such attempt should display suitable error messages.

10. DOWNLOAD <connection id 1> <file1> <connection id 2> <file2> <connection id 3> <file3>

This will download a file from each host specified in the command. Please note that the total number of hosts may be 1, 2 or 3. The hosts will be part of the list of hosts displayed by the LIST command.

E.g., if a command DOWNLOAD 2 file1 3 file2 4 file3 is entered to a process running on the server UNDERGROUND, then the process running on UNDERGROUND receives this command, it will

request file1 from embankment, file2 from highgate and file3 euston. All these files are to be sent parallelly using TCP connections. The local machine will automatically accept the file and save it in the same directory where your program is under the original name. When the download is complete this process will display a message indicating so. Also, the remote machine will display a message in its user interface indicating that a file (e.g. a.txt, b.txt or c.txt) has been downloaded along with the hostname of the host from which the file was downloaded. Upon completion, a message is displayed. If the download fails for some reason, an error message should be displayed. When a download is occurring, a message should be displayed on the local machine. If the download fails for some reason, an error message should be displayed on the remote machine.

Similarly as what UPLOAD requires, at the end of each successful download, you will print Rx rate at which the Receiver downloaded those files and the Tx rates at which each Sender sent those files.

NOTE: Both notifications and data sending should use TCP connections. Make sure the clients send their files in parallel even though they cannot start sending files exactly at the same time.

NOTE: DOWNLOAD command on a server should display an error message. No files should be downloaded from the server.

11. CREATOR Display your (the students) full name, your UBIT name and UB email address.

4. Analyze

Make sure you test your program for some values of file sizes between 1000 Bytes and 10 MBytes, and then for sizes {50, 70, 100, 150, 200} MBytes. You should also test your program for different packet sizes (or the size of the buffer you read from the file and send at a time using the send socket call), ranging from 100 Bytes to 1400 Bytes. These two parameters will be referred to as *File Size* and *Packet Size*. You can generate files of different sizes using the UNIX utility *dd*. For example, to generate a file of size 512 bytes, use the command:

dd if=/dev/zero of=test_file_to_create count=1

Here count=1 refers to 1 block of 512 bytes.

NOTE: Please delete these test files once you have tested your application and do not include any of these in your submission.

4.1 Data Rates vs. File Size

Run your application for some value of *File Size* between 1000 Bytes and 10 MBytes, and then for sizes {50, 70, 100, 150, 200} MBytes. Observe the Tx Rate and Rx Rates. Keep the *Packet Size* to be

constant at 1000 Bytes and do a single file transfer at any time for these measurements.

Write down your observations. What variations did you expect for data rates by changing the file size and why? Do they agree with your measurements; if not then why? Remember to analyze the entire system, which includes not only the network, but also the Tx and Rx ends.

4.2 Data Rates vs. Packet Size

Run your application for different *Packet Sizes* (ranging from 100 Bytes to 1400 Bytes, by increasing in steps of 200 Bytes), and observe the Tx Rate and Rx Rates. Keep the file size constant at 200 MBytes and do a single file transfer at any time for these measurements. Write down your observations. What variations did you expect for data rates by changing the packet size and why? Do they agree with your measurements; if not then why? Remember to analyze the entire system, which includes not only the network, but also the Tx and Rx ends.

4.3 Data Rates vs. Load Variations

Run your application on different machines, and start multiple file transfers over the network at the same time. Vary the load over the network, by varying the number of simultaneous file transfers from 1 through 3, i.e., {1,2,3}. Keep the File Size constant at 70 Mbytes and Packet Size at 1000 Bytes for all transfers.

Write down your observations. What variations did you expect for data rates by changing the load and why? Do they agree with your measurements; if not then why? Remember to analyze the entire system, which includes not only the network, but also the Tx and Rx ends.

4.4 Install iperf and Measure Network Bandwidth

Iperf is a tool for measuring network performance, especially the maximum TCP/UDP bandwidth available. For more information about the tool, refer to its homepage.

Install iperf :

1. On a CSE student sever (e.g., timberlake) download iperf source, by using this command:

    wget http://sourceforge.net/projects/iperf/files/iperf-2.0.5.tar.gz

2. Untar the source and examine its README.

3. Execute ./configure

4. make (Do not go for the make install step)

5. Execute the iperf binary (i.e., executable) present in the src subdirectory as follows:

On the server:
timberlake {~/iperf-2.0.5/src} > ./iperf -s

Then on the client:
embankment {~/iperf-2.0.5/src} > ./iperf –c timberlake.cse.buffalo.edu

Note: To measure the available bandwidth, you run iperf on one machine which acts as a server and another which acts as a client. The client sends packets (TCP/UDP) to the server as fast as it can, and measures the network bandwidth. For more information regarding this, refer to iperf help.

To execute iperf to measure the network bandwidth available between two machines say timerlake (server) and embankment (client), run the iperf binary as follows(you may add TCP port number which is not showed in this example):

Analysis: Observe the available bandwidth which iperf outputs for 1 client and then increase the number of clients from 1 through 3, by running the iperf on other client machines also as you did on embankment above.

Compare the results (available bandwidth) of iperf for a single client and for multiple clients (load variations) with the results obtained from your File Sharing Application (section 4.3). Justify why do/don't they conform to each other. Briefly record all your analysis and observations for sections 4.1, 4.2, 4.3 and 4.4 in a file called Analysis-A1.txt or Analysis-A1.pdf and turn in with your implementation.

5. Submission and Grading

5.1 What to Submit

Your submission should contain a tar file – Name it as < your ubit_name>.tar:

- All source files including Makefile (Name your main program as proj1.c)
- Your analysis for sections 4.1, 4.2, 4.3 and 4.4 in a file named Analysis-A1.txt/pdf.
  (Usage of graphs for your analysis is highly recommended.).

5.2 How to submit

Use the submission command submit_cse589 to submit the tar file.

5.3 Grading Criteria

- Correctness of output
- Your analysis

- Organization and documentation of your code

5.4 Important Key Points:

- *There is just one program. DON'T submit separate programs for client and server.*
- *All commands are case-insensitive.*
- *Error Handling is very important – Appropriate messages should be displayed when something goes bad.*
- *DON'T ASSUME. If you have any doubts in project description ask the TA's.*
- *NO multiple submissions please. If you have more than 1 submission will not consider any of your submissions.*
- *Submission deadline is hard. No extension. Any submission after 23:59 of Feb28st 2014 will be considered as late submission.*
- *Please do not submit any binaries or object files or any test files.*