CSE 589 Fall 2015
Programming Assignment 1
Distributed File Sharing System
Due Time: 10/16/2015 @ 23:59:59

1. Objective

Getting Started: Familiarize yourself with socket programming.
Implement: Develop a simple application for distributed 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 for measuring network performance.

2. Getting Started

2.1 Socket Programming - Beej Socket Guide: http://beej.us/guide/bgnet

3. Implementation

3.1 Programming environment

You will write C (or C++) code that compiles under the GCC (GNU Compiler Collection) environment. If you use any other language apart from C or C++ your project will not be graded. Furthermore, you should ensure that your code compiles and operates correctly on the CSE student servers listed. This means that your code should properly compile by 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 allowed to use any STLs for the socket programming part. If you use any STLs for socket programming, your 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: ./prog1 s 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: Server should always be run on timberlake and no other host involved in file transfer should be run on timberlake. Use other CSE student servers to run hosts involved in file transfer.

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.

Servers to be used:

Timberlake (only to run server. Don't do any file exchange on timberlake).
Following servers can be used for file-exchange:
Euston (euston.cse.buffalo.edu)
Embankment(embankment.cse.buffalo.edu)
Underground (underground.cse.buffalo.edu)
Highgate (highgate.cse.buffalo.edu)

NOTE: You should use only the servers listed above. Don't use any other department servers. Also, create your own directory in /local/Fall_2015/ on all the hosts except Timberlake where you can use your home directory. Use only this directory to store/run your programs. Also, make sure you have set appropriate permissions to your folder so that it cannot be accessed by any other students.

NOTE: Please don't use any other directory/CSE servers to run your code.

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 command options.

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

3. DISPLAY: Display the IP address of this process, and the port on which this process is listening for incoming connections.

NOTE: The IP should not be your "Lo" address (127.0.0.1). It should be the actual IP of the host.

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 other peers currently registered with the server. The first task of every host 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 host registers or a registered host exits, the server should update its Server-IP-List appropriately and then send this updated list to all the registered clients. Hosts 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 host 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. If the host closes the TCP connection with the server for any reason then that host should be removed from the "Server-IP-List" and the server should promptly inform all the remaining hosts.

NOTE: The REGISTER command works only on the client and not on the server. Registered clients should always maintain a live TCP connection with the server.

5. CONNECT <destination> <port no>: This command is used to establish a connection between two registered clients. The 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 timberlake.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 host by the server. Any attempt to connect to an invalid IP or an IP address not listed by the server in its Server-IP-List should be rejected and suitable error message 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.

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.

id:	Hostname	IP address	Port No.
1:	timberlake.cse.buffalo.edu	192.168.21.20	4545
2:	highgate.cse.buffalo.edu	192.168.21.21	5454
3:	underground.cse.buffalo.edu	192.168.21.22	5000
4:	embankment.cse.buffalo.edu	192.168.21.22	5005
5:	euston.cse.buffalo.edu	192.168.21.22	5125

NOTE: The connection id 1 should always be your server running on timberlake.cse.buffalo.edu. The remaining connections should be the peers whom you have 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 highgate 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. QUIT Close all connections and terminate this process. When a host exits, the server unregisters the host and sends the updated “Server-IP-List” to all the clients. Other hosts on receiving the updated list from the server should display the updated list.

9. GET <connection id> <file> This command will download a file from one host specified in the command.

E.g., if a command GET 2 file1 is entered for a process running on underground, then this process will request file1 from highgate. 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 completes 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) has been downloaded along with the hostname of the host from which the file was downloaded. Upon completion, a success message is 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 and local machine, e.g., the file that your local machine tries to download doesn't exist. Also the peer should serve files located in your own directory to requesting peers. The peer should not serve files located in any other directory.

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

10. PUT <connection id> <file name> For example, ‘PUT 3 /local/Fall_2015/qiao/a.txt’ will put the file a.txt which is located in /local/Fall_2015/qiao/, to the host on the connection that has connection id 3. An error message is displayed if the file was inaccessible or if 3 does not represent a valid connection or this file doesn't exist. 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. Also other peers should not be able to monopolize your resources, e.g., another peer should not be able to fill up this peer's disk.

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

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.

11. SYNC This command will make sure that all peers are up-to-date with their CONNECTED peers. For example, peer A has a file A.txt; peer B has a file B.txt; peer C has a file C.txt; Peer A, B, and C are connected with each other. After executing this command on any peer, all three peers should have file A.txt, B.txt and C.txt under its directory. For another example, peer A has a file A.txt and is connected with peer B only; peer B has a file B.txt and is connected with peer A and C; peer C has a file C.txt and is connected with B only. After synchronization, peer A should have A.txt and B.txt; peer B should have A.txt, B.txt, and C.txt; peer C should have B.txt and C.txt. For this command, you need to design and implement a strategy such that file synchronization can be done in a parallel manner, i.e., all peers should start synchronization in a parallel way and each peer should also download/upload files from/to its connected peers in a parallel way. To make it simple, we assume each peer only needs to get a txt file from each connected peer named by this connected peer’s name. For example, if highgate is connected with euston and embankment, then highgate will only need to get “euston.txt” from euston and “embankment.txt” from embankment; the other files will be neglected. The server timberlake does NOT need to participate file synchronization task. Therefore, file synchronization will only be carried out among euston, highgate, embankment, and underground. Each peer should print out an event message when starting and finishing synchronizing a file, following the format “Start/End XXX.txt at HH:MM:SS”, where HH, MM, and SS represent the hour, minute, and second of local time correspondingly”. For example, when embankment is executing synchronization with euston and highgate, we may see the following printout in embankment’s terminal:

> Start euston.txt at 10:21:30
> Start highgate.txt at 10:21:31
> End euston.txt at 10:21:40
> End highgate.txt at 10:21:42

It is worth mentioning that each peer (including timberlake) should be able to accept SYNC command and trigger synchronization of the other peers.

NOTE: You CANNOT use timberlake server as a file relay but you can use it notify/schedule the other peers.

4. Analysis

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 sizes {50, 70, 100, 150} MBytes. Observe the Tx Rate and Rx Rates. Keep the packet size to be constant at 1000 Bytes. Do 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 150 MBytes and does 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.

NOTE: For all the above analysis you decide on the file-chunk size. The selected file-chunk size should be meaningful and should aid your analysis. Don't forget to specify the file chunk size you have used in your analysis. Use different file chunk size and don't keep it constant for all the analysis experiments. You don’t need to analyze synchronization task.

5. Submission and Grading

5.1 What to Submit?

Your submission should contain a tar file – Name it as <ubit_name>.tar: All source files including Makefile (Name your main program as ubit_name_proj1.c).

Your design document named <ubit_name_design.pdf>which in details describes your design rational for function 11 in section 3.3, how you prevent other peers from filling up your disk and accessing your file outside your own directory in your implementation.

Your analysis for sections 4.1 and 4.2 in a file named <ubit_name_analysis.pdf> (Usage of graphs for your analysis is highly recommended. Please don't write huge texts) .

If you want to get bonus points, submit another document named <ubit_name_extra.pdf> to describe in detail your design and implementation on how you improve the performance. Analysis and results are required.

5.2 How to submit

Use the submission command submit_cse589 to submit the tar file.

5.3 Grading Criteria

5 points each for function 1-8.
10 points each for function 9 and 10.
20 points for function 11
10 points for design document
10 points for analysis
15 bonus points for extra design, implementation and analysis
Total is 100+15.

5.4 Important Key Points

1. There is just one program. DON'T submit separate programs for client and server.
2. All commands are case-insensitive.
3. Error Handling is very important – Appropriate messages should be displayed when something goes bad.
4. DON'T ASSUME. If you have any doubts in project description ask the TA's.
5. Please try to avoid multiple submissions. If you have any multiple submissions we will consider the latest submission before the deadline.
6. Submission deadline is hard. No extension. Any submission after 23:59:59 of Oct 16th 2015 will be considered as late submission.
7. Please do not submit any binaries or object files or any test files.