CS536 - Fall 2014
Programming Assignment 1
Remote File Sharing System

Due Date: Tuesday, September 23rd, 2014 before 11:59 PM Eastern Time

The deadline is a hard deadline and no extensions will be granted. The deadline is 11:59 **Eastern time**. Make sure to check what your local time will be.

I. Objectives

In this programming assignment, you will implement and analyze a peer-to-peer file sharing application.

- *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.

II. Getting Started

A. Socket Programming

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

B. POSIX Threads

PThreads Tutorial: http://www.llnl.gov/computing/tutorials/pthreads

C. iperf

iperf Homepage: https://iperf.fr/

III. Implement

A. Programming environment

You must work individually on this assignment; we will use MOSS (http://theory.stanford.edu/∼aiken/moss/) to check for plagiarism.

You will write C++ (or C) code that compiles under the GCC (GNU Compiler Collection) environment. You should ensure that your code compiles and operates correctly on the XINU machines (xinu1.cs.purdue.edu, xinu2.cs.purdue.edu, etc.) found in HAAS 257. 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 XINU machines and should function correctly when executed.

B. Running your program

Your process (your program when it is running in memory) will take 1 command line parameter that corresponds to the port on which your process will listen for incoming connections. The program should be named `rfss` (i.e., you can run it as: `./rfss 4322` where `4322` is the port number).

C. Functionality of your program

Use TCP sockets for your implementation. When launched, your process will listen for incoming connections on the specified port. Meanwhile, your process will provide a user interface (running under a separate thread) that will offer the following command options:

Note: Specific examples are used below for clarity.

1) ***HELP****:* Display information about the available user interface options.

2) ***EXIT****:* Close all connections and terminate this process.

3) ***MYIP****:* Display the IP address of this machine.

4) ***MYPORT****:* Display the port on which this process is listening for incoming connections.

5) ***CONNECT host_name port_number****:* Establish a new connection to the machine with the specified name that is listening on the specified port number (e.g., 3456). After using this command, the user interface on this process will remain unavailable until a success message or failure message is displayed. Whenever a remote process tries to connect to this process, the connection (if possible) is set up automatically and a message is displayed. You should not allow more than one connection with a particular remote host (i.e., only one connection between any pair of IP addresses). You should not allow connections to yourself. Each process should never maintain more than N total connections. #define N to 5 in your program.

6) ***LIST****:* Display a numbered list of all the connections that this process is part of. This numbered list will include connections initiated by this process and connections initiated by other processes. The list includes the name and IP address of the other end of each connection. For example, the output should be in the same format as

1 : xinu2.cs.purdue.edu : 128.10.3.52
2 : xinu3.cs.purdue.edu : 128.10.3.53

7) ***TERMINATE connection_ID****:* This will terminate the connection listed under number 2 when LIST is used to display all connections. In this example, the connection with xinu3 should be terminated. An error message is displayed if there is no valid connection with the given number. If a remote machine terminates one of your connections, you should also display a message.

8) ***GENERATE file_name file_size****:* This will generate a file which is located under the local path `Upload/` with the specified name and size (contents do not matter). Note that you can generate files of different sizes using the UNIX utility **dd**. For example, to generate a file of size 512 bytes, you can use the command

dd if=/dev/zero of=*filename* count=1

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

9) ***UPLOAD connection_ID upload_file_name****:* This will upload the file named **upload_file_name** which is located under the local path `Upload/` to the host on the connection that is designated by the number 4 when LIST is used. An error message is displayed if the file was inaccessible or if 4 does not represent a valid connection. The remote machine will automatically accept the file and save it in its local path `Download/` under the original name. When the upload is complete, this process will display a

message indicating so. Further, the remote machine will display a message in its user interface indicating that a file has been downloaded and the file name is given. 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, print the rate at which the Transmitter (Sender/Uploader) uploaded the file. Similarly, print at the receiver, the rate at which the file was received. We will call these *Tx Rate* and *Rx Rate*, respectively. 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 xinu1 to xinu2, the format for printing this information is:

At Tx end:
Tx (xinu1): xinu1 -> xinu2, File Size: x Bytes, Time Taken: y useconds, Tx Rate: z bits/second.
At Rx end:
Rx (xinu2): xinu1 -> xinu2, File Size: a Bytes, Time Taken: b useconds, Rx Rate: c bits/second.

Note: You must use *packet size*-byte buffers and send the buffer contents using the send socket call until the entire file has been transmitted.

To make it easy to change the packet size in your experiments, read the packet size value from a configuration file named `config.txt` located in the same directory as your program.

You can use the 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.

10) ***CREATOR****:* Display your (the student's) full name and Purdue email address.

11) ***Bonus: DOWNLOAD file_name****:* Many peer-to-peer systems like BitTorrent choose the best host to download from based on end-to-end measurements. For bonus points, you can implement a download function that queries each of the open connections for a file named **file_name**. Among those that reply positively (indicating that they have a file with the specified name), it selects the one that replies with the shortest delay, and downloads the file from that connection into its `Download/` directory. To make the delays different, either use machines not in the XINU lab for some of your connections, or create higher load on some of the machines. If you complete this part and are interested in a more sophisticated download implementation, you can implement a BitTorrent-like mechanism that concurrently downloads different chunks of the file from multiple peers.

IV. ANALYZE

Make sure you test your program, `rfss`, for a few values of file sizes between 1000 Bytes and 10 MBytes, and then for sizes {20, 30, 40, 50, 60, 70} 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*, respectively. **Note that these experiments may be time-consuming, so start early!**

Please delete the test files once you have tested your application and do not include any of them in your submission, so as not to exhaust our limited disk quota.

A. Data Rates vs. File Size

Run your application for a few values of file size between 1000 Bytes and 10 MBytes, and then for sizes {20, 30, 40, 50, 60, 70} MBytes. Observe the Tx Rate and Rx Rates. Keep the packet size constant at 1000 Bytes and do a single file transfer at a 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.

B. 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 50 MBytes and do a single file transfer at a 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.

C. 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 N. For $N = 5$ that would be {1,2,3,4,5}. Keep the File Size constant at *50 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 two end systems.

D. 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 the iperf homepage.

Install iperf:

1) While logged onto a XINU machine, download iperf by using this command:
   wget https://iperf.fr/download/iperf_2.0.2/iperf_2.0.2-4_i386.tar.gz
2) Untar the downloaded archive and examine its README file.
3) Execute **./iperf –help** to become familiar with iperf.
4) Execute the iperf binary (i.e., executable) present in the **src** subdirectory as follows.

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 iperf, refer to iperf's homepage.

To execute iperf to measure the network bandwidth available between two machines say xinu1 (server) and xinu2 (client), run the iperf binary as follows:

Server : xinu1> iperf -s
Client : xinu2> iperf -c xinu1

E. Analysis

Observe the available bandwidth which iperf outputs for 1 client and then increase the number of clients from 1 through $N = 5$, by running iperf on other client machines also as you did on xinu2 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 IV-C). If the results are similar, explain why; if they are different, explain why.

V. Submission and Grading

A. What to Submit

Your submission directory should contain contain:

- All source files.
- A README.txt file describing how to compile and execute your file sharing application. (We recommend using a makefile to compile your files. Your executable must be named `rfss`) Please document any reasonable assumptions you are making in this file, e.g., maximum file name length, and indicate if any parts of your assignment are incomplete, and if you have implemented the bonus.
- Your analysis for section IV in a file named *Analysis-A1.pdf*.

Please do not submit any binaries or object files or transfer test files.

B. How to submit

Note that the deadline is specified using Eastern Time. You will need to submit one hour earlier (your local time) if you are on Central time, two hours earlier if on Mountain time, and three hours earlier if on Pacific time.

Submit every time you have a stable version of any part of the functionality. Make sure to submit early to make sure you do not miss the deadline due to any last minute congestion.

1) Login or ssh to a XINU lab machine, e.g., xinu1.cs.purdue.edu
2) In the parent directory of your submission directory, type the command:

   turnin -c cs536 -p pa1 <submission-dir-name>

   where <submission-dir-name> is the name of the directory containing your files to be submitted. For example, if your program is in a directory `/homes/abc/assignment1/src`, make sure you `cd` to the directory `/homes/abc/assignment1` and type:

   turnin -c cs536 -p pa1 src
3) If you wish to, you can verify your submission by typing the following command:

   turnin -v -c cs536 -p pa1

   Do not forget the **-v** above, as otherwise your earlier submission will be erased (it is overwritten by a blank submission).

   Note that resubmitting overwrites any earlier submission and erases any record of the date/time of any such earlier submission.

   We will check that the submission timestamp is **before the due date**; we will not accept your submission if its timestamp is after the due date, even by 2 minutes. Do NOT submit after 11:59 PM Eastern Time.

*Questions about the assignment should be posted on **Piazza**. If you want to include details of your design or a code snippet from your program, do not post to Piazza but email the TA at **tsai46@cs.purdue.edu**. The TA may discuss with the professor if necessary.*