CS3251 Computer Networks I

Fall 2012

Programming Assignment 1

Assigned: Sept. 13, 2012

Due: Sept. 27, 2012, 11:59pm

PLEASE READ THIS CAREFULLY BEFORE YOUR PROCEED

For this assignment you will design and write your own application program using network sockets. You will implement your application using TCP (SOCK_STREAM) sockets.

The application you are designing and implementing is a somewhat simplified version of the classic original Napster peer to peer file sharing system. Before you start you should read this description of how old Napster worked (http://www.howstuffworks.com/napster.htm).

There are two main components of the system.

1- Content listing, search and peer discovery at the servers, and
2- Peer-to-peer file sharing

In this first assignment you will design and implement the first component. We will add the second component in the second assignment. While you will be graded on how these commands work to implement component 1 only, you may want to anticipate what will be needed to add component 2 in your design.

You will build a Napster-Classic server that keeps a listing of available files at peers that connect to it. With each file the server should keep the IP address of the peer holding this file. If there are multiple peers that maintain the same file (same name), these should be shown as separate listings with different IP addresses. To simplify life you can assume that file names are single words with a size limit (make your own limit).

You will also implement clients (or Peers) who can make two types of requests.

a) "addfile" command: will add a file listing to the server. A switch on this command (something like –d) will cause the file to be deleted. Only the file associated with this IP address should be deleted. Ideally you should be able to add multiple file names in one command.

b) "filelist" command: will cause the server to send a listing of all the files available on all peers it has heard of up until it received the filelist command.

You may implement this in a "non-persistent" manner. That is, after a command completes the socket is closed. The next command will require a new socket and TCP connection. You may implement this as an iterative server. The server, of course, has to continue to run. The server application should be designed so that it will never exit as a result of a client user action.

You will need to develop your own "protocol" for the communication between the client and the server. While you should use TCP to transfer messages back and forth, you must determine exactly what messages will be sent, what they mean and when they will be sent (syntax, semantics and timing). Be sure to document your protocol completely in the program write-up.

Focus on limited functionality that is fully implemented. Practice defensive programming as you parse the data arriving from the other end. Do not work on a fancy GUI, but focus on the protocol and data exchange. Make sure that you deal gracefully with whatever you or the TA might throw at it.

You may also use client and server templates including the ones discussed in class. However, **you must include a citation (text or web site) in your source code and project description if you use an external reference or existing code templates.**

Notes:

1. Your programs are to be written in standard C using the standard Socket libraries we have been covering in class. Your programs should run on the Unix (or Linux) systems.
2. You should work individually on this assignment. While you may discuss various aspects of sockets programming with each other, you may not re-use each other’s code.
3. You should select a port for your service. I recommend something between say 13000 and 14000. It is wise to provide the server port as input to the server and not to hard code the server port in the program. Note that if your server crashes, its port number may not be usable for a short period of time afterwards (for reasons we will explain later) so it is best to choose a new port number when you restart it.
4. Use explicit IP addresses (in dotted decimal notation) in the client for specifying which server to contact. Do not use DNS for host name lookups.
5. Make sure that you do sufficient error handling such that a user can't crash your server. For instance, what will you do if a user provides invalid input?
6. Your server will be iterative; handling one client at a time. Nevertheless it should not be bothered if multiple clients attempt to access it simultaneously. You should test scenarios with “race conditions” where multiple adds and/or lists are requested at the same time.
7. You must test your program and convince us it works! Provide us with a sample output that shows all of your functions working.
8. If you feel there is any ambiguity in the system description above, feel free to add your own specification as long as it does not trivialize the problem.

Submission Format:

Submit your programming assignments as a gzipped tarball (.tar.gz extension) with your Last Name as the file name. *You submit everything using t-square.*

Your archive(.tar.gz) should include: your **well-documented source code**, a README file, a Makefile and sample output file called Sample.txt. The README file must contain:

1. Detailed instructions for compiling and running your client and server programs including a Makefile.
2. A description of your application protocol (1/2 to 1 page) with sufficient detail such that somebody else could implement your protocol
3. Any known bugs, limitations of your design or program

Grading Guidelines:

We will use the following guidelines in grading:

0% - Code is not submitted or code submitted shows no attempt to program the functions required for this assignment.

25% - Code is well-documented and shows genuine attempt to program required functions but does not compile properly. The protocol documentation is adequate.

50% - Code is well-documented and shows genuine attempt to program required functions. The protocol documentation is complete. The code does compile properly but fails to run properly (crashes, or does not handle properly formatted input or does not give the correct output).

75% - Code is well-documented. The protocol documentation is complete. The code compiles and runs correctly with properly formatted input. But the program fails to behave gracefully when there are user-input errors.

100% - Code is well-documented. The protocol documentation is complete. The code compiles and runs correctly with properly formatted input. The program is totally resilient to input or network errors.