COP 4635 SYS & NET II – UDP CLIENT-SERVER COMMUNICATION

OVERVIEW

This assignment will continue with simple client-server network programming. This time, you will create several programs that will communicate with each other over a network using UDP as the underlying network protocol. A server program will be designed to accept requests from clients and respond to those requests. Client programs will make requests of the server process and display the returned results. You are required to implement two different client programs, one in C and one in Java to test client-server communication.

THE PROGRAM

When completed, your server program (`UDPserver.c`) will perform the following tasks:

1. Create a UDP socket given a port number that is accepted from the program's argument list as a single integer. Display host and port information on the screen.

2. Wait for incoming requests from client programs and respond to them in an endless loop. When a request arrives, the server will
   a) record the location of the client machine on the screen and interpret the request that was sent by parsing the message body.
   b) Depending on the message body, the server will perform the following actions:
      c) If the message body starts with *<echo>* and ends with </echo>, the server will read the text message between *<echo>* and </echo> that it received and return it as a reply in a new message body that stars with *<reply>*, followed by the received text message and ends with *</reply>*. There should be no additional whitespaces introduced by the server between <reply> and </reply> and the text message received by the client.
      d) If the message body is *<loadavg/>*, the server will compute its load average for 1, 5, and 15 minutes using the library function `getloadavg()`. Values returned by the function represent the number of processes in the system run queue averaged over the last 1, 5, 15 minutes. The three numbers must be represented as a string with each value separated by a ":". The string should start with <replyLoadAvg> and end with </replyLoadAvg>. In the return message, there should be no whitespaces between <replyLoadAvg> and </replyLoadAvg> and the actual encoded values for the load average.
      e) If the message body is <shutdown/>, the server will shut down gracefully by closing the socket and then terminating.
      f) In all other cases, when the message is not formatted as described above, the server responds with an error message in the following format: <error>unknown format</error>.
3. The main server will continually loop, processing requests as they come in. The server will gracefully shutdown if it receives a <shutdown/> message from a client.

The server must handle messages according to the protocol above in a robust manner. This means that the server needs to respond to the requests correctly and handle incorrectly formatted messages gracefully without crashing

or failing to respond. You may assume that the total size of the message exchanged between server and client is fixed and exceeds no more than 256 bytes. The message includes message body, message header, and message footer. Please note that message body and footer may be empty for some of the requests.

When completed, your client programs (`UDPclient.c` and `UDPclient.java`) provide functions to perform the following tasks:

1. Create a datagram socket.
2. Send a request for service to the server.
3. Receive a response from the server.
4. Close the socket.

Use the provided header and stub files uploaded to the Content area in *eLearning* to implement the functionality of the clients. For C, write the testing code in a separate main program `UDPmain.c`.. Because we will use a separate Java and C main program to evaluate your solution, it is important that you comply with the interface definitions of the clients as provided by the header and Java stub files. Your `UDPmain.c` program and `main()` in Java should execute the following sequence of steps:

1. Accept a hostname and a port number from the command line in the format "`<hostname> <portnum>`" where `<hostname>` is the name of a host, and `<portnum>` is a port number.
2. Create a socket and connect to the server at the specified location.
3. Send a request to the server (user entered or a fixed message).
4. Receive a reply from the server and display it in its entire length.
5. Close the socket and terminate the program.

The client must call the corresponding function implemented in `UDPclient.c` to connect, send, receive messages, and close the socket.

IMPLEMENTATION SUGGESTIONS

I suggest reading back through the lecture notes for details on each step in completing this project. You should also consider examining the reading material. Read these pages thoroughly before starting this project. For your implementation you need to use the following system calls for the C client-server program and the corresponding library calls for the Java client program:

socket()	**gethostbyname()**	**gethostname()**	**bind()**
sendto()	**recvfrom()**	**close()**	**getloadavg()**
getsockname()			

DELIVERABLES & EVALUATION

Submit your complete solution as a single zip file (only zip files will be accepted) containing A) source code, B) a makefile to compile the client and server programs, and C) a README file (if applicable) to the corresponding dropbox in *eLearning*.

The README file should only be included if you submit a partial solution. In that case, the README file must describe the work you did complete. You must follow the *Project Submission Instructions* posted in *eLearning* under Course Materials. The submission requirements are part of the grading for this assignment. If you do not follow the requirement, 5 points will be deducted from your project grade. Keep in mind that documentation of

source code is an essential part of programming. If you do not include comments in your source code, points will be deducted. We also require you to refactor your code to make it more manageable and to avoid memory leaks. Points will be deducted if you don't refactor your code or if we encounter memory leaks in your program.

Your solution needs to compile and run in the computing environment provided on the CS department's Linux servers. Graders will upload your solution to a server and compile and test it running several undisclosed test cases. Therefore, to receive full credit for your work it is highly recommend that you test & evaluate your solution on the servers to make sure that graders will be able to run your programs and test them. You may use any of the 5 SSH servers available to you for programming and testing and evaluation. Use *ssh.cs.uwf.edu* to log into any of the 5 servers.

GRADING

This project is worth 100 points in total. A grade sheet posted in *eLearning* outlines the distribution of the points and grading criteria. Keep in mind that there will be a substantial deduction if your code does not compile.

DUE DATE

The project is due on February 13th by 3:00 pm before the start of the class. Late submissions will not be accepted and the grader will not accept any emailed solutions. The syllabus contains details in regards to late submissions.