Project – Socket Programming

This project is actually pretty simple … if you have some background in software development using "bare metal" tools. By "bare metal" I mean the use of an independent text editor, command-line compiling and linking, and possibly makefiles. If you're more comfortable using an integrated development environment (IDE) of your choice, that's fine too.

There are examples of TCP client/server and UDP client/server code listed in the text book. We've gone over much of the code, and it should work pretty much "out of the box" … perhaps with a few small tweaks to the include files, depending on your platform and choice of development environment. A zipfile is attached with my slightly tweaked client/server code, as well as some makefile examples that we discussed in class.

For the platform, I heartily suggest some flavor of Unix (eg. Linux, such as Ubuntu, Fedora, Slackware, etc. or MacOS) rather than Windows. For the development environment, I suggest a simple GNU toolchain such as "gcc" for the compiler/linker, "nedit" or "gedit" for the text editor, and "make" to simplify the compile/link steps. If you choose to use an integrated development environment (IDE), that's fine, but make sure you understand where the interesting widgets are hidden.

Your assignment is to type in these programs, compile/link them, and make them run … then make a few small modifications to them to show that you know what's going on. The tweaks you'll be making will create a sort of "modified echo server" that accepts a packet with a client sequence in it, and then responds with an extended sequence.

Refer to the example client and server code for more details on the software implementation. Also, in developing/testing your client/server software, be sure to leverage virtual machines or multiple physical machines to make sure your implementation functions correctly on a network (virtual or physical) where client and server are not using a communication path that is purely internal to a single system.

Deliverable #1:

You have functional TCP and UDP client/server programs able to be compiled and linked, and you have "gone above and beyond" just typing these things in and making them run. In other words, you've modified one set of the programs to …

- reach into the socket data structures on the server (sockaddr),
- retrieve the IP address and port of the client (in_addr),
- translate and display the client data locally on the server (ntoa, printf),
- send data back to the client as text in the packet payload (write or sendto),
- retrieve the transmitted data from the client's socket buffer (rbuf), and
- display it locally on the client (printf).

This is not difficult. It takes about 2 lines of modification to the server code to get the socket data and write it. Ask questions if you need help. Or Google. There are lots of examples of this sort of thing on the Internet.

Deliverable #2:

Modify the provided client and server code to implement a simple protocol between the communicating nodes, such as:

- The client …
  - obtains one or more numbers or letters from the user
  - counts how many numbers/letters are obtained (N)
  - inserts this count (N) and the beginning index (j) into the payload of a packet along with the text of the numbers/letters obtained
  - sends the packet to the server
  - waits for a response
- The server …
  - waits for a client connection, and upon sensing it, does the following:
  - retrieves the client's IP address from the socket & displays it locally
  - retrieves the count and index from the incoming PDU, and displays them locally
  - checks the received sequence for errors
  - creates an ACK packet in the same format as the received packet, except...
    - for **correct sequences**, the ACK packet must contain an “error” field indicating no errors, and the index of the response must start at j+N+1
    - for **incorrect sequences**, the ACK packet must contain an “error” field indicating errors, as well as the corrected original sequence
  - sends the ACK packet to the client
- The client …
  - receives the ACK packet, checks it for errors & responds appropriately
  - if the sequence is **correct**, the extended sequence should be displayed locally
  - if the sequence is **incorrect**, the data should be displayed in a format that highlights the error, and the user should be given another chance to enter a correct sequence

In all cases, your system should print to the local display everything that is happening at the level of the application protocol. In other words, if the client sends “1,2,3” or “a,b,c,d” the client's display should show what it is currently transmitting, and what it receives in response from the server (ie. “4,5,6” or “e,f,g,h”). Correspondingly, the server's display should show what it has received and is currently transmitting.

For example:

- if the client obtains from the user a sequence of 3 numbers starting at 4,
  - the packet sent from client to server would contain: 3 | 4 | “4,5,6”
  - the packet sent from server to client would contain: OK | 3 | 7 | “7,8,9”
  - in this case, the “OK” at the beginning of the server's response indicates that the client's transmission was a correct sequence
- if the client obtains from the user a sequence of 4 numbers such as 1,2,3,5
  - the packet sent from client to server would contain: 4 | 1 | “1,2,3,5”
  - the packet sent from server to client would contain: KO | 4 | 1 | “1,2,3,4”
  - in this case, the “KO” at the beginning of the server's response indicates that the client's original transmission was an incorrect sequence

Deliverable #3:

You must create a protocol diagram for this “modified echo” protocol along with pictures showing the definition & structure of the application-layer packets exchanged between the client and server.

Refer to the protocol diagram on TRACS that we've used before for an example. The protocol diagram is like the state diagrams of the client and server processes unwrapped along a common time axis, showing state transitions, inputs/outputs, and so on. Yours doesn't need to be as detailed as the one on TRACS, but something along these lines will help in understanding what is going on.

The picture showing structure of protocol packets should be similar to those used in your earlier presentations regarding various Internet protocols (eg. boxes with a discussion of contents and length).

What you should submit:

1. Your source files (TCP and/or UDP) in C, with any/all custom headers files, and
2. Your protocol diagram / protocol definition diagram in PDF.

Bonus Points:

1. Make a LUA dissector for your protocol similar to what we did for the packet capture of SDDS and show me its performance in Wireshark.