CS 422 (Park) **Assignment V** **Due: Apr. 11 (Mon.), 2011**

Submission instructions: Please type your answers and submit electronic copies using `turnin` *by 4pm on the due date. You may use any number of word processing software (e.g., Framemaker, Word, LaTeX), but the final output must be in pdf format that uses standard fonts (a practical test is to check if the pdf file prints on a CS Department printer). For experiments and programming assignments that involve output to terminal, please use* `script` *to record the output and submit the output file. Use* `gnuplot` *to plot graphs.*

PROBLEM 1

For students who have the 5th edition: Read chapters 20–26. For students who have the 4th edition: Read chapters 16–18, 20–26.

PROBLEM 2 (40 pts)

As a continuation of Problem 2, Assignment IV, run the TCP remote command client/server application of Problem 3, Assignment IV, between a pair of machines in LWSN B158 with dedicated point-to-point Ethernet links and sniff all the TCP traffic for a client/server transaction at the client side using `tcpdump`. How many IP packets are involved in the client/server transaction? Separate them by direction: from client to server and server to client. For the first and last IP packet in the packet trace, show a hexadecimal dump of all its header fields. How big is the IP header? Is fragmentation activated by IP? How big is the TTL value? What does the TOS field specify? Is the value meaningful? How is the type of IP's payload (in this case TCP) indicated? Are the number of TCP segments different from the number of IP packets? For the first, second, middle (one carrying application payload somewhere in the middle), and last TCP segment, show a hexadecimal dump of all the header fields. What are the sequence numbers of the first two TCP segments? What TCP header flag bits in the 4 captured segments are set, and what is their meaning? What are the window size values of the first two segments? Show the hexadecimal dump of the application payload of the middle TCP segment. Decode and match the payload content with the terminal output seen at the client machine.

PROBLEM 3 (70 + 40 + 40 pts)

(a) Under /u/u3/park/pub/cs422, you will find `talk_client.c` and `talk_server.c`, which are simplified implementations of `talk`, a precursor of today's messaging applications. From a programming technique perspective, the main feature of interest in `talk` is that you be able to see (i.e., read) what the other party is writing to you in the midst of your own write. This means that when you write—i.e., the program reads the characters you type on the keyboard—packets arriving from the other party carrying messages must be handled asynchronously and displayed on the terminal. Hence, a solution where write followed by read is repeated in an infinite loop is inadequate. To achieve asynchronous I/O, a SIGPOLL (or SIGIO) signal handler must be written and registered with the kernel as in Problem 4, Assignment IV.

The client/server talk example is incomplete in several respects. First, the syntax for turning a file descriptor into asynchronous mode uses System V UNIX (e.g., Solaris) convention, which must be modified to match the syntax suited for Linux. This is one of the OS dependent aspects of network programming that must be carefully handled when porting code across different UNIX platform. Second, the SIGPOLL handler is missing. Writing a SIGPOLL handler is straightforward but more important is an application design decision: how to display a newly arriving message on the same terminal so that what you are writing is not scrambled with what you are receiving. Using the `curses` library, one can split the terminal in two so that what you write appears on the top half and what you read appears on the bottom. By using windowing platforms such as X Window System, the same can be accomplished by demuxing write/read across two windows. Third, `talk` is by its very nature a peer-to-peer (P2P) application where there is no distinction between client and server: they are one and the same.

Modify and complete the provided code and get it to work by solving the three problems. Do not use `curses` nor separate X Window System primitives or widgets to separate input/output. (We are not practicing GUI programming, nor are we concerned with pretty output. But we are concerned about legible output.) Invent a rudimentary

but working solution using *stdio*. For example, if in the middle of typing a response a new message arrives, you may display the new message on a new line, and on the following line, refresh the interrupted message so that one can continue typing where one left off. Remove any client/server dependencies in the code so that the same code, call it `my_talk`, is used at both parties. The convention for using `my_talk` should be

```
% my_talk hostname port-number
```

where `hostname` and `port-number` are the coordinates of the party you wish to talk to. The port number is optional so that if not specified then the messaging app uses a default port number of your choice. Test your talk application on two hosts—open two shells, one of them remote—and record the interaction using `script`. In this instance, you are talking to yourself. At the start, make sure that `my_talk` is running on both hosts before initiating actual text messaging. This is to prevent potential system crash resulting from the other party not being online when you type a message. These and other clean-up issues must be dealt with separately. Finally, find another student or friend and test your P2P messaging app on two PCs, one in LWSN B148 and the other in LWSN B158. Record the output using `script`.

(b) Extend the P2P talk app such that multi-party messaging—up to 4 users/peers—is supported. Test your application on four hosts on your own by opening four shell windows—three remote—and recording the messaging interaction using `script`. Test the application by finding two class mates (or friends) with whom 3-way messaging is demonstrated using your multi-party messaging application on three PCs.

(c) An extension of part (a), implement a keep-alive feature where a peer transmits a special `keep-alive` control message to its counterpart periodically—every X seconds where X is a configurable system parameter—to inform that it is alive. The periodic transmission of `keep-alive` should be implemented using a signal handler for SIGALRM. Control data should be transmitted over a separate UDP socket. In tandem, a peer should check if a `keep-alive` message has arrived every second time (i.e., $2\,X$ seconds) that it transmits a `keep-alive` to its counterpart. This may be done using a counter mechanism and flag. If a `keep-alive` message has not arrived, the app terminates with a suitable message. Test the keep-alive feature using X values of 1, 2, 10 and 30 seconds and terminating one of the two peers during a messaging session.

PROBLEM 4 (70 pts)

A continuation of Problem 3, Assignment IV, modify the TCP-based remote command client/server application so that the client runs on Windows (XP or later releases). Since the server structure is specific to Linux/UNIX, it remains the same: a remote command server for Linux. Porting the client code entails using WinSock, Microsoft's system call implementation of UNIX socket calls. Most features remain the same with some modifications stemming from Windows idiosyncrasies. A key difference is that Windows does not support a signal abstraction in its kernel as in UNIX but Problem 3 does not use signals. For development, you may use any of the Windows machines in CS Windows labs (use your CS Windows account), ITaP operated PCs (Purdue account), or your own PC/laptop if equipped with a C compiler. Visual C++ is a popular programming environment but not necessary. For benchmarking, use a command-line window in Windows (do not run the client from Visual C++ directly during benchmarking) to run the client process. The server runs on one of the machines in LWSN B148. Use secure shell to open a remote window to the Linux lab. Perform the benchmarks as in Problem 3, Assignment IV. Use graphical screen capture (in place of `script`) to log the terminal interaction the client side.