CS 422 (Park) **Assignment V** **Due: Nov. 7 (Wed.), 2007**

Submission instructions: Please type your answers and submit electronic copies using `turnin` *by 5pm 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 or ps format that uses standard fonts (a practical test is to check if the pdf/ps file prints on a CS Department printer without problem). 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. Use* `ps2gif` *to convert a eps/ps plot to gif format (e.g., for inclusion in Word) if there is a need.*

PROBLEM 1

Read Chapters 13, 16–21, 24 and 26 from Comer.

PROBLEM 2 (30 pts)

As a continuation of the `tcpdump` traffic measurement and sniffing problems of Assignment IV, utilize the trace of Problem 5, Assignment IV, and additionally capture the UDP client/server session of Problem 4, Assignment IV, using `tcpdump`. Your trace should log all the packets involved in the UDP-based client/server application. In the TCP trace, do a hexadecimal dump of the IP and TCP headers (you need to check if IP or TCP are invoking any optional header fields) of the Ethernet packets leaving the PC where `tcpdump` is run, and the same goes for the Ethernet frames arriving at the PC, that are engaged in the three-way connection start-up handshake. Additionally do a hexa dump of the packet(s) carrying the response from the server. Decode all the IP and TCP header fields of the above identified packets. Is fragmentation going on in the captured packets? Is the TOS field value meaningful? How big is the TTL value? How big are the IP and TCP header sizes of the captured packets? What is the first sequence number used in the TCP header field? Do the ACK numbers in the TCP header field match any of the sequence numbers in TCP packets traveling in the opposite direction? What is the initial window size used by TCP? (Varies between operating systems.) How many Ethernet packets, in total, are involved in carrying out the UDP-based client/server interaction? How many are involved in the TCP-based client/server interaction? How do they compare with respect to overhead? Last, but not least, identify in the TCP session trace the three-way connection tear-down packets and decode their header values. Using the time stamps provided by `tcpdump`, calculate how long it took from the moment of the first tear-down (i.e., FIN) packet and until the last tear-down packet.

PROBLEM 3 (60 pts)

Under ~park/pub/cs422, you will find `talk_client` and `talk_server`, which are simplified implementations of `talk`, a precursor of today's messenger 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 code 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 SIGIO (or SIGPOLL) signal handler must be written and registered with the kernel, which is a callback function that the kernel invokes whenever a new packet arrives. When working with signals, it is good practice to test that a signal handler is working properly in isolation (e.g., by sending 10 packets and checking that the signal handler at the receiving machine is invoked 10 times) before the signal handling component is integrated into the rest of the system.

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 SIGIO handler is missing. Writing a SIGIO 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 X-windows, 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 separation of client from server. They are one and the same.

Complete the provided code and get it to work by solving the three problems. When registering your SIGIO handler, use `sigaction` in place of `signal`. Do not use `curses` nor separate X-windows 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 IP-address port-number
```

where `IP-address` and `port-number` are the coordinates of the party you wish to talk to. 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 in class who has a correctly running program and test that your P2P talk applications work correctly together. Record the output using `script`—you should be using your fellow student's `talk` application, and vice versa—and indicate in your submission which `talk` version you like better. Try to be objective.

PROJECT PROBLEM

This is the first of two project problems which counts toward 1/3 of the project score. **The due date of the project problem is Nov. 14 (Wed.), 2007.** A variant of Problem 4, Assignment IV, implement the simplified iterative UDP-based client/server application so that it runs on Windows XP (or Vista). This means that the code has to be ported so that it uses WinSock, Microsoft's system call implementation of UNIX BSD socket calls. Most features remain the same, with some modifications stemming from Windows idiosyncrasies (not specific to XP). You only need port the server application. For development, you may use your laptop (if it runs Windows) or any of the Windows machines available in CS Windows labs (use your CS Windows account). Visual C++ is a popular programming environment but not necessary. For benchmarking, use a command-line window in Windows XP (do not run the server from Visual C++ directly during benchmarking) to run the server process. Use secure shell to open a remote window to the Linux lab. Run the client process on a Linux machine (since they talk TCP/IP, it doesn't matter from which OS the client and server processes are run). Perform the same benchmarks as in Problem 4, Assignment IV. Use screen capture to record the interaction and output in the two windows. Submit the code (and any compilation instructions or scripts) and output.