CS 422 (Park) **Assignment III** **Due: Feb. 27 (Mon.), 2006**

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 (20 pts)

Read Chapters 8, 9, from Comer. Solve Problem 8.3 where "your site" is interpreted to mean our lab (REC 108). Draw a network diagram showing how the various PCs and laptops are connected to each other. Consult the TA to find out how the machines in REC 108 connect to the outside world. Use a drawing tool of your choice (if you aren't sure, check with the TA).

PROBLEM 2 (30 pts)

The purpose of this problem is to introduce "sniffing" raw data from the "wire" on the private LAN in the lab consisting of two PCs, two laptops, and Ethernet hub. Execute the following command on a PC to capture Ethernet frames from the network:

```
% sudo /usr/local/etc/tcpdumpwrap-eth1 -c16 -wEX
```

The command will capture 16 packets from the Ethernet LAN and save them in the file `/var/tmp/`*login-EX*, where *login* is your login ID. Generate your own traffic on the private network by doing `ping` to communicate with a machine on the private net (`ping -c count 10.0.0.X`) where `count` is the number of packets transmitted. What is the value of `count` that you need to set so that `tcpdumpwrap` captures 16 packets? Submit the log file in hexadecimal format. Also, save your `ping` terminal interaction using `script` and submit it along with the hexadecimal `tcpdumpwrap` log file. Using the Ethernet header format(s) discussed in class, decode from the hexadecimal dump what the values for the fields in the Ethernet header are. Compare these values across the 16 captured frames. Use `/sbin/ifconfig` to compare the Ethernet addresses that you have snooped with those printed out by `ifconfig`. From the value of the type/length field determine whether the Ethernet frames are DIX- or IEEE 802.3-compliant. Explain how you are able to distinguish between the two. What is the default payload size of `ping`? How long is the payload of the Ethernet frame? Are they consistent? (The default IP header size is 20 bytes.) Noting that the first four bits of an IP header indicate its version number (4 or 6), locate the version number bits in the Ethernet payload and identify the IP version running on your test machine. Noting that the last 8 bytes of a 20-byte IP header represent the 4-byte IP source address and 4-byte IP destination address, respectively, locate the IP source/destination address bits in the payload and compare their values to those output by `ifconfig`.

PROBLEM 3 (50 = 40 + 10 pts)

(a) As a continuation of Problem 5, Assignment II, extend the client/server application such that any Linux command can be requested by the client to be executed by the server and its output returned to the client. The request format should be of the form

$$\$\ command\ \$\ argument\text{-}1\ \$\ argument\text{-}2\ \$\ \cdots\ \$\ argument\text{-}n\ \$$$

where *argument-1, argument-2, . . ., argument-n* represent command-line arguments (possibly empty), and "$" is a delimiter symbol. On the server side, after a child is forked, it sleeps for 3 seconds before it executes the requested command. Also, remove the `waitpid()` system call. On the client side, before sending the command request, print the current time on the terminal. After receiving the server's response, print the current time. Test your program by first running the server application in the background, then executing `client.bin` multiple times (open multiple shell windows) with `date`, `uptime`, `who`, `ps u`, `ls -l`, `echo clientrequestisthis`, `sleep 5`, `ls -a -c -d -i -l -q -r -t -u -l`, `ping -c 5 xinuserver`, and `terve`. Use `script` to record the run-time session. Submit your code and output. You must provide adequate documentation in your code. (*Remark: Make use of string processing library functions to minimize the parsing effort.*)

(b) What is the effect of deleting `waitpid()` in the server code? Since `waitpid()` is a blocking call and you don't want the server to block on `waitpid()`—there could be other client requests waiting in the FIFO queue—what can you do to reap the benefit of `waitpid()` without incurring its cost? Use `gnuplot` to compare the response times of the 11 client requests where *reponse time* of a client request is the difference in the time stamps before sending the request and after receiving its response. Use `time date`, `time uptime`, . . ., `time terve` to determine the wall clock time that these commands take when run directly through a shell. Using `gnuplot` compare these numbers against the response time numbers gotten in the FIFO-based client/server application. Estimate the overhead (%) introduced by client/server interaction. In your evaluation, is the overhead small or large?

PROBLEM 4 (30 pts)

Extend the client code in Problem 3(a) so that it sets an alarm (i.e., signal SIGALRM) of 1.010 seconds before sending out the command request (use `ualarm()` for that). If the response does not return before the alarm expires, the request is retransmitted to the server. If it arrives prior, the alarm is cancelled. To handle SIGALRM asynchronously, you need to register a SIGALRM signal handler using `sigaction()` at the start of your client code. The server code does not change. Benchmark your "improved" client on the test cases in Problem 3(a). Use `script` to record the run-time session. Discuss the differences (if any) that you see when comparing with the `script` output in Problem 3(a) of the old client.

PROBLEM 5 (30 pts)

Suppose that you are sending a file of size $\mathcal{S}$ bytes using a simplified form of sliding window control over a point-to-point link of bandwidth $\mathcal{B}$ (bps) and propagation delay $\mathcal{L}$ (sec). On the sender side, a window (or block) of k packets is transmitted back-to-back, each packet carrying a payload of X bytes. The receiver, upon receiving k packets belonging to the i'th block, sends an ACK packet to the sender of size $\mathcal{A}$ bytes. The sender, before transmitting a block, sets a timer T, and retransmits the block of packets if an ACK does not arrive before T expires. When the ACK arrives, the next block of packets is transmitted. Assuming the link is noise-free (no bit flips occur), what is the perfect value for T? Derive an expression for T. Based on this best T, derive the *completion time* C formula: the time from the moment the first bit of the file hits the wire on the sender side to the moment the last bit of the file is received at the receiver. Finally, derive the utilization ϱ of the link assuming you are the only user of the transcontinental gigabit link. Supposing that you're sending a 100 MB file over a point-to-point link from NYC to London over a 1 Gbps link burried on the ocean floor of the Atlantic (estimate the propagation delay L by looking up the distance from NYC to London and dividing it by the speed of light), what are the optimal timeout value T and completion time C when X is 1000 bytes, $\mathcal{A}$ is 100 bytes, and $k = 1$ (aka stop-and-wait)? What is link utilization ϱ? What are their values when $k = 10$? To bring the completion time down to 5 seconds, what block size must you use with what timeout T? What is the resultant utilization ϱ? Recalculate the above when you're sending the file from Purdue to a certain university in Bloomington, IN (again, estimate propagation delay by dividing distance by speed-of-light) with $k = 10$. Does propagation latency make a significant difference on file transfer time performance? Discuss your results.