CS 422 (Park) **Assignment III** **Due: Oct. 5 (Fri.), 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 (20 pts)

(a) The stop-and-wait protocol, a special case of ARQ, only sends the next data frame after the current frame has been acknowledged by the receiver. Despite this "one frame at a time" feature, why is it necessary to have a 1-bit sequence number as part of the frame header? As part of your answer, describe an example scenario that shows what can go wrong if no sequence numbers are used.

(b) Stop-and-wait's throughput has a particularly simple formula, *frame size* $\div$ RTT, which holds true when all frames—data and ACK—are correctly received. How does the throughput formula change if a data frame is not correctly transmitted with probability p? You may assume independence, that is, the probability that two successive data transmissions fail is p^2, three successive failures is p^3, and so forth. You may assume that ACK frames are always correctly transmitted. What happens to the throughput formula if ACK frames also fail transmission with independent probability s? Note that communication errors on the Internet are not symmetric.

PROBLEM 2 (25 pts)

Perform a traceroute experiment to `www.ucla.edu` to determine the approximate RTT from our lab at Purdue. Assuming a bandwidth of 10 Mbps from UCLA to Purdue, if a server (i.e., sender) at UCLA were to use stop-and-wait with frame size 1500 bytes (you may ignore overhead introduced by header bytes) to transmit a 10 MB file, how long would it take to complete the transfer assuming no frame transmission failures? How long is the expected file transfer completion time if a data frame transmission from UCLA to Purdue fails with probability $p = 0.01$ (i.e., 1% failure rate). What if $p = 0.001$? Download a large file from `www.ucla.edu` (do some browsing on their web site) and time the approximate completion time using a watch. How does the actual file transfer completion time compare to that of stop-and-wait (without errors)? Repeat the performance evaluation for `www.umich.edu`.

PROBLEM 3 (50 pts)

As a continuation of Problem 5, Assignment II, modify the server side such the server's child process, before calling `exec`(), sleeps for 7 seconds. Modify the client side so that when no response is forthcoming from the server within S seconds after the request has been sent, a duplicate request is retransmitted. This retransmission is repeated every S seconds until the server's response is received or the number of attempts has exceeded 5. To implement request retransmission at client side, use the signal SIGALRM that is set using `alarm`() with S as argument. The alarm should be set right after transmitted a request. Register a signal handler, `my_retransmit_req`(), using the `sigaction`() system call. `my_retransmit_req`() is a callback function that you are registering with the kernel such that it is invoked when SIGALRM is triggered (i.e., "raised" in UNIX jargon). Your signal handler should retransmit the request, check that the number of attempts has not exceeded 5 (don't hardcore the constants in your code but use constant definitions in the program header for modularity), then call `alarm`() to set a new alarm. Test your client/server application with $S = 10$, 5, 2, and 1. Use `script` the record the interaction and output.

PROBLEM 4 (50 pts)

Modify the concurrent client/server application in Problem 5, Assignment II, so that the server becomes a file server. That is, the client sends the name of a file, *file-name*, instead of a command, which the server searches for in its current directory. If the file does not exist, an error status (the string "*file-name* not found") is returned. Simplify

the server so that it becomes an iterative server, i.e., it does not fork a child process to handle the actual request after parsing but processes the request itself (the server is unithreaded). The client/server application uses stop-and-wait to transmit the contents of the requested file. That is, the server process sends a packet of X bytes to the client process (except for the last packet which could be less than X bytes) through a separate FIFO that is used to transmit payload from server to client (use a similar naming convention as in the original request FIFO when setting up the data FIFO). You may assume reliable channels so that retransmission (for now) using timeout does not need to be implemented. Nonetheless, because it is a stop-and-wait protocol, the client must send an ACK packet through the request FIFO. The server waits for the ACK of the data frame it has sent before transmitting the next data frame. Note that if a data or ACK frame were ever to be lost, the client/server application would hang forever. The parameter X should be specified as a command-line argument when the server process is run. The client should print out the file transfer completion time at the end of run by taking a time stamp (using `getttimeofday`()) after sending the request, taking a time stamp after receiving a special end-of-file signaling frame (use an empty frame), and outputting their difference (in units of seconds). Benchmark your application with the file /u/u9/park/pub/cs422/test-shannon.txt for $X = 1024$, 2048, and 8192 bytes.