CS 422 (Park) **Assignment VI** **Due: Dec. 6 (Mon.), 1999**

PROBLEM 1 (20 pts)

As a continuation of Problem 3, Assignment V, make the modification such that the sender—instead of transmitting packets with fixed, *deterministic* interarrival times or packet spacings—uses an exponential distribution as its "oracle" to set the packet spacing between successive packets. The code for generating random deviates from an exponential distribution with parameter λ, i.e., $f(x) = \lambda e^{-\lambda x}$, is given by the function `exponential_dist()` available on the course homepage. A random variable obeying an exponential distribution with parameter λ has mean $1/\lambda$. Hence, if you are asked to perform an experiment with an *average* interpacket arrival time of 0.5 sec, then you have to call `exponential_dist()` with the argument $\lambda = 1/0.5$ to receive a value which has the proper meaning.

Repeat the experiments carried out in Problem 3, Assignment V, for a buffer size of 50, plot the packet drop and buffer occupancy traces, as before. Compare the plots with your previous plots for each value of average interarrival time. Give an interpretation of your results.

PROBLEM 2 (50 pts)

Implement a "greedy" file transport protocol called `ftp-greedy` which achieves fast (or not-so-fast), reliable file transport on top of UDP using a form of selective ARQ. Given a file F of size $|F|$ bytes to be transferred, `ftp-greedy` segments F into $\lceil |F|/M \rceil$ blocks of size $M < |F|$ (except for possibly the last block which can be of size less than M), sending each block encapsulated in an UDP packet. Before encapulation, each block is assigned a header where a sequence number i identifying the block is enscribed. The sequence number is represented using a single byte, thus giving a range of 0–255. If more than 255 blocks are needed, i mod 255 is used to achieve wrap-around. `ftp-greedy`, on the sender side, is to be invoked using

% `ftp-greedy` *filename IP-address*

where *filename* is the name of the file to be transferred and *IP-address* is the IP address in dotted decimal form. On the receiver side, `ftp-greedy` is executed without command-line arguments. This puts `ftp-greedy` in "receive mode." When the transfer of a file is complete, `ftp-greedy` (on the receiver side) prints out a message to *stdout* saying that the file in question has been (reliably) received after which it terminates. For the receiver to know the filename in which to store the received file, the sender, in block 0, sends the filename. To indicate that no more blocks are to come (i.e., the last block i has been sent), the sender sends an empty block with sequence number $i + 1$ mod 255 at the very end.

Two parameters (e.g., hardcoded using #define) influence the sender's behavior, M and S. M is the block size and S is the sender's interpacket waiting time (in msec). That is, S measures the number of milliseconds the sender waits between successive sending of packets (an event triggered by SIGALRM). The smaller S, the more "greedier" `ftp-greedy`. The receiver uses negative ACKs to achieve reliable communication using retransmission. As soon as the receiver sees a "jump," i.e., a hole in its received packet sequence,

$$\cdots i-2,\, i-1,\, i,\, i+2$$

it transmits a NACK packet to the sender demanding that packet $i + 1$ be retransmitted. At the same time, it starts a timer to expire after T msec. If the missing packet is received prior to the timer expiration, the timer is cancelled. If not, a NACK is sent again. Set T initially to 3 msec; each time a NACK is sent again because of timer expiration, increment T by 2 msec. Each time a retransmitted packet arrives before timer expiration, decrement T by 1 msec. T should never be allowed to go below 3 msec or above 15 msec. The receiver can maintain a buffer size of K packets to hold packets arriving while the missing $i + 1$ packet is being processed. You can maintain the buffer anyway you want, but it can never exceed the K packet bound.

Test your application by creating a $|F| = 100$ KB dummy file containing the ASCII characters '0', '1', '2', ..., '9', '0', '1', ... Measure the file transmission completion time (wall time) at the receiver from the moment that the first packet is received to the moment that the last packet is received. The last packet may be marked by sending a termination packet with empty payload. Perform the measurements for $S = 10, 30, 50, 70, 100, 200$ msec with $M = 1000$ B. Draw the performance curve (completion time) as a function of S.

PROBLEM 3 (50 pts)

The receiver or server in Problem 3, Assignment V, can be viewed as a *router*. Upon dequeueing a packet from its user space buffer, if, in addition, a routing table look-up and other processing is performed to determine where to *forward* the packet to—of course, the UDP packet must contain the IP address of its ultimate destination inside its payload (a technique called *tunneling*)—then this is precisely the function that a router carries out.

Extend Problem 3, Assignment V, in the following way. First, turn the server into a router which forwards the UDP packet to the destination IP address contained in the first four bytes of its payload. This entails making a small change to the SIGALRM handler to handle forwarding. Otherwise, everything else remains the same. Second, modify the *same* server of Problem 3, Assignment V—now in a slightly different way—such that it becomes a bona fide *receiver* in the end-to-end congestion control sense discussed in class. In particular, use SIGPOLL to increment a counter called *throughput* which counts the number of packets that arrived within a specific time interval *history*. Use SIGALRM, which is set to go off every *history* milliseconds, to transmit to the sender—for this one, send it directly to the sender (note, its IP address is enscribed in the first four bytes of the payload received) by-passing the router—the value contained in *throughput* (e.g., as a 4-byte integer). Log the *throughput* value with timestamp into a LOG file. Reset *throughput* before returning from the SIGALRM handler. Third, modify the sender of Problem 3, Assignment V, such that it transmits packets to the router where the first four bytes contain the destination's IP address, as required, but more importantly, it adjusts the interarrival time or packet spacing based on the user's *desired-throughput* and the actual throughput conveyed by the receiver in the feedback packet. In particular, if the actual throughput is greater than the desired throughput by more than 10%, then the interarrival time is increased (thus decreasing the sending rate) by some fixed value—a parameter $\varepsilon > 0$ which is expressed in milliseconds. If actual throughput is smaller than *desired-throughput*, then then interarrival time is decreased by ε. Before each packet transmission, log into a file the value 1/interarrival time with its time stamp. The user's target throughput, adjustment parameter ε, the router's IP address, and the receiver's IP address are to be given as command-line inputs.

Set the router's interservice time (period between SIGARLMs) to 50msec and its buffer to 50. Set the user's target throughput to 10 packets per 500msec. Thus the receiver's time window, *history*, should take on the same value 500msec. Hardcode the initial interarrival time as 300msec. Set ε to 50msec and run the experiment for a duration of 120 seconds. Plot the graphs for the log files generated at the sender and receiver. How does your system perform with respect to achieving the user's desired throughput efficiently? Give your interpretation. Repeat the experiment with ε set to 5, 20, 100, 300 milliseconds. Plot the corresponding graphs and give your interpretation of the observed results.