CS 536 (Park) **Assignment VIII** **Due: Nov. 23 (Mon.), 1998**

PROBLEM 1

Read "Congestion avoidance and control" by Van Jacobson (Proc. ACM SIGCOMM '88, pp. 314–329, 1988). Give a 1-page summary/critique of the paper. Comment on the main pros/cons of the paper from your perspective.

PROBLEM 2

(a) Implement an end-to-end sender/receiver application pair using UDP where the sender process `udp-send` on the source host transmits a sequence of UDP packets to the receiver process `udp-receive` on the destination host. `udp-send` takes on two command-line arguments

% `udp-send` *interarrival-time session-duration*

where *interarrival-time* is the mean (i.e., $1/\lambda$) of an exponential interarrival time distribution with rate λ (measured in milliseconds) and *session-duration* is the total session or flow duration (i.e., after *session-duration* seconds from start-up, `udp-send` is to terminate), measured in seconds. The receiver, `udp-receive`, takes on a single command-line argument *service-time* which is the mean (i.e., $1/\mu$) of an exponentially distributed service time distribution.

The receiver works by responding to two signals—SIGPOLL and SIGALRM—in an asynchronous manner using signal handlers. When a SIGPOLL interrupt occurs, the receiver's SIGPOLL signal handler fetches the newly arrived UDP packet(s) from the kernel's buffer space and enqueues the packet into its own (i.e., user space) buffer, a FIFO queue. For tracing purposes, the SIGPOLL handler also takes a time stamp and records both the time stamp and the current queue length (excluding the newly arrived packet) into a LOG file for queue length called *log-queue* for later examination. If the user space buffer, at the time of packet arrival, is full, then the UDP packet is dropped or discarded. This event is then also recorded using a time stamp into a separate LOG file for packet drops called *log-drops*.

When a SIGALRM interrupt occurs (signalling that a packet should be serviced), the receiver's SIGALRM signal handler dequeues a UDP packet from its FIFO queue. Before exiting the handler, the exponential distribution function—`exponential_dist` (see the accompanying function code in the course homepage)—is called with parameter λ to determine when to wake up next.

Note: An important programming consideration here is the management of the two interrupts such that they do not get in "each other's way." For example, while the SIGALRM signal handler is active, the SIGPOLL signal should not be able to interrupt the SIGALRM signal handler—and vice versa—as this can lead to a corrupted state of the user space buffer which is manipulated by both. As part of the hand-in, explain in half a page of write-up how your program manages the orderly access to the user space FIFO queue by the two signal handlers.

Test your application using the following set-up. Use a fixed MTU (for UDP packet payload) size of 1kB (containing any bit pattern) and a user space buffer capacity of 500 MTUs (i.e., 500kB). For mean service time $1/\mu$ = 200ms, run six experiments with $1/\lambda$ = 300ms, 260ms, 220ms, 200ms, 180ms, 160ms, for a fixed session duration of 30s. For each run, plot the time series for the queue length and packet drops and compute their mean. Give your interpretation of the queueing behavior as λ is increased.

(b) As a continuation of part (a), how does the queueing behavior seen in part (a) for the regime when $1/\lambda$ = 300ms, 260ms, 220ms (and, perhaps, 200ms) compare to the steady-state queue length formula predicted by M/M/1 queueing analysis? Does the latter fit—and can you make it fit with some adjustment (plot their respective graphs)—to the actual observed queueing bahavior? What are some of the intrinsic factors that make the quantitative prediction given by the M/M/1 formula deviate or differ from actual measurements?

Repeat the experiments of part (a), now, with buffer capacities 20kB, 15kB, 10kB, 5kB, and 1kB. Plot the packet drop rate, one, as a function of buffer capacity, and two, as a function of interarrival time for the five buffer capacities (put the five graphs into a single plot). Give your interpretation of the packet drop behavior under Poisson traffic for the given system configurations.

Repeat the experiments of part (a), now, with a fixed *deterministic* service time of $1/\mu$. Compare the time series plots for queue length and packet drops with the case when the service time was exponentially distributed. What differences do you observe? Give an interpretation of the observed results.

PROBLEM 3

Derive the steady-state queue length distribution $X_n(\infty)$ for the finite queue length (or buffer capacity) case of N, $n \leq N$. Can you relate the finite queue length formulae to the infinite queue length formulae (in particular, when do they approach each other)?

Comparing the results (queue length and packet drops) predicted by the M/M/1/N queueing analysis and the measured performance results obtained from part (b) of Problem 2 for buffer capacities 20kB, 15kB, 10kB, 5kB, and 1kB, how accurate or valid are the quantitative values predicted by analysis? How about the qualitative (i.e., shape of the plots) behavior predicted by analysis? Justify your conclusions.

PROBLEM 4

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 used as

% `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 example,

% `ftp-greedy`

filename received (*X bytes*)

%

might be the interaction seen on the terminal. 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. For the S corresponding to the smallest completion time, call it S^*, test your application for $M = 500, 1500, 2500$ B. Draw the performance curve as a function of M. Give a 1/2-page written interpretation of your results. Hand in the source code, scripts, and your summarized results.