CS 536 (Park) **Assignment IV** **Due: Dec. 6 (Thu.), 2001**

PROBLEM 1 (30 pts)

The purpose of this problem is to perform "snooping" of raw data from the "wire" using promiscuous mode on the private LAN consisting of Xinu machines, laptops, connected by an Ethernet hub in REC108. We will examine the format and content of the captured frames. Execute the following command on a Xinu machine in REC108 to capture a packet from the private network:

% `/usr/local/etc/sudo /usr/local/etc/snoopwrap-elxl1 -o filename -c1`

The command will read a packet from the Ethernet LAN and save it in the file `/var/tmp/username-filename`, where `username` is your login name and `filename` is the string specified with the `-o` option. You will have to generate your own traffic on the private network by pinging a machine on the private net (`/usr/sbin/ping 10.0.0.X`), by sending a UDP packet (e.g., using the UDP based programs in one of the earlier homework assignments), and by using telnet to communicate with a machine on the private net (`telnet 10.0.0.X`). Execute the three cases and print out the log files in octal format. In each case, give a precise explanation of the captured frame, its format and content, including the various header fields, their meaning and values. For example, in the case of `telnet`, the captured Ethernet frame will encapsulate an IP datagram, which, in turn, will encapsulate a TCP segment. Thus three separate headers need to be explicated. For a description of the ICMP fields, confer the relevant RFC available, e.g., at `http://www.ietf.org/rfc.html`.

PROBLEM 2 (30 pts)

(a) Use `/usr/local/etc/traceroute` to determine the routes taken by IP packets on their journey to a specific IP destination address. Use server addresses available in `http://www.traceroute.org` to discover routes to destinations of your choice, from your Xinu machine to three IP addresses in each of the following regions: midwest (US), east coast (US), west coast (US), South America, Asia, Europe, Australia/New Zealand, and Africa (if you can find a site). For each region, tabulate the average number of hops taken as well as the minimum and maximum, how many hops are common to the three destinations within each region, their delay characteristics, if there are any hops along a path who do not generate ICMP packets when the TTL reaches zero (ICMP TIME_EXCEEDED response)—is this possible?—and any other traits you find noteworthy. For each region, determine, from the hop names, how many distinct autonomous systems (AS) are traversed on the way to the destinations, including how many routers within each AS. Are you able to find regions/IP addresses for which the forward path from your Xinu machine to the target is different from the reverse path taken from the destination to the Xinu machine? What is the longest hop count you were able to find? **Important: Do not write scripts to carry out the traceroutes. Perform them manually. Only send at most 5 traceroutes to the same destination IP address within a single day.**

(b) For one destination address from each region, use `/usr/sbin/ping` with packet size 64 bytes (default), inter-packet spacing 2 seconds, and 30 probe packets total to determine 30 sample RTT values. For each region, plot a histogram of the observed RTT values where the X-axis (abscissa) denotes RTT and the Y-axis (ordinate) denotes the number of samples with that RTT value. The granularity of RTT is 1 millisecond. Perform the same experiment with the destination IP address being one of the Xinu machines on the same LAN. For each case, compute the mean and standard deviation of the RTTs. From the shape of the histograms, what can you conclude with respect to the RTT estimation problem? Have your probes encountered any losses? How does that impact the RTT estimation problem? **Important: As with part (a), perform the measurements manually and not via a script. Make sure to abide by the 2 second spacing and 30 probe packet limit so as to not adversely affect the target site and trigger false alarms.**

PROBLEM 3 (80 pts)

Implement an adaptive congestion control algorithm in a sender-router-receiver set-up which utilizes the help of the router as follows. The router maintains a buffer of size B wherein incoming UDP packets are enqueued. It dequeues packets with service rate μ (i.e., service interval $1/\mu$). This part is an extension of Problem 4 in Assignment III. The dequeued packet is forwarded to a destination whose IP address and port number are specified in the first 6 bytes of the UDP packet payload (IP address followed by port number). So that the destination may know who the real sender is, rather than spoofing the source IP address, the router will inscribe into the first 6 bytes of the forwarding packet the sender's IP address and port number, overwriting the information that was put in by the sender. This is an application of the source routing problem tackled in Problem 3 of Assignment II.

The router sends a control packet to the sender each time it awakes to service its queue indicating what the current queue length Q, $0 \leq Q \leq B$, is. Thus the control packet carries a field where the length information is inscribed. The sender implements a rate-based congestion control algorithm whereby an interpacket spacing parameter Λ (msec) is used to separate successive packets in time. This is an infinite source model and the larger Λ the smaller the mean packet generation rate. The sender is to execute the following control law:

$$\text{new } \Lambda \leftarrow \text{old } \Lambda + \epsilon(Q - Q^*)$$

where $0 \leq Q^* \leq B$ is a target queue length that is to be maintained. Here $\epsilon > 0$ is an adjustment factor. Thus, if $Q > Q^*$, then Λ is increased which is tantamount to decreasing the sending rate, and vice versa.

Test the algorithm using $\mu = 20$ such that the mean time between successive servicing of packets is $1/\mu = 50$ msec; set $B = 40$ (packets of fixed MTU size of 1 kB), $Q^* = 20$, and set the initial value of Λ as $1/(2\mu)$. Run the algorithm five times using five different values for ϵ, from "very small" to "very large." For each run, record the size the of queue at each dequeueing instance into an array (do not use `printf` since this will slow the router down unnecessarily) and plot the trace at the end of the run. Each run should consist of the generation of 1000 UDP packets. Note that the receiver here is a purely dummy entity. Give an interpretation of your data as a function of varying the parameter ϵ. Is there an optimal value for ϵ? As a criterion of "goodness," for each benchmark run, calculate the average of the absolute difference of the actual queue lengths (logged by the router) from the target queue length Q^*. Thus, 0 is perfect, and values larger than 0 indicate worsening performance with respect to achieving the target.

For your best value of ϵ, repeat the same experiment for 2 simultaneous senders, 4 senders, and 6 senders. What differences do you observe? Give an interpretation of your results.

PROBLEM 4 (60 pts)

Assume the router can only use a single bit to communicate to the sender about its queue state: "0" means too short and "1" means too long. (Note that there is no specific value for "just right.") Design and implement your own controller—submit a single page write-up explaining what your controller does and why you chose this design—that reacts to the binary form of feedback. Benchmark your algorithm on the $1/\mu = 50$ msec, $B = 40$, $Q^* = 20$ set-up employed in Problem 3 for 1, 2, 4, and 6 senders. What performance differences do you observe when compared with the more sophisticated setting of Problem 3? Give an interpretation of your results.

PROBLEM 5 (100 pts)

In this problem, we will combine congestion control with NACK-based packet recovery to build a pseudo real-time audio player with QoS amplification. The audio player consists of a client and a server, `myaudioc` and `myaudios`, respectively, and the server's packets—which are directed at the client—are sent through a router, `myrouter`, which forwards UDP packets using a SIGPOLL signal handler (no SIGALRM handler and no additional buffers) to the client. As in Problem 3, the router will assume that the first 6 bytes of the payload comprise the client's destination IP address and port number. The router is run with 2 command-line arguments

% `myrouter` *drop-prob rand-seed*

where *drop-prob* is a probability with which it will drop incoming packets (use `random()` modulo 100,000), and *rand-seed* is a seed to be used when initializing the pseudo-random number generator using `srandom()`.

The sender, i.e., server, follows the same general structure as the sender in Problem 3 and is executed as

% `myaudios` *router-IP router-port client-IP client-port audio-file packet-size epsilon*

where *router-IP* and *router-port* are the router's coordinates, similarly *client-IP* and *client-port* for the client, *audio-file* is an audio file in .au format (use ~park/pub/cs536/pp.au under xinuserver), *packet-size* is the segment size by which *audio-file* is split up and transmitted (by default 1 kB), and *epsilon* is the best ϵ found in Problem 3. The sender formats UDP packets where the first 6 bytes of the payload comprise the client's IP address and destination port number, the next 4 bytes form a sequence number (starting at 0 and incremented for every *packet-size* data sent), followed by the actual audio data. The sender/server receives three types of feedback packets from the receiver (i.e., client), a "congestion control" (CC) type which is indicated by the first byte in the payload having value 0 and the next 4 bytes encoding the client's buffer state $Q - Q^*$ (triggering the same update procedure as in Problem 3), a "negative ACK" (NA) type which is indicated by the first byte in the payload having value 1 and the next 4 bytes containing a sequence number of a missing packet, and a "positive ACK" (PA) type which is indicated by the first byte in the payload having value 2 and the next 4 bytes containing a sequence number which represents the cumulative ACK up to the indicated sequence number. The sender maintains a circular buffer of 100 *packet-size* units, which is flushed only when a PA type feedback packet is received. Along with freeing up the circular buffer triggered by the cumulative ACK received, audio data are read in from *audio-file* to fill up the available buffer space. When a NA-type feedback packet is received, the requested packet is retransmitted using the copy resident in the circular buffer.

The client/receiver follows the overall structure of the router in Problem 3 with a couple of augmentations. It has the command-line arguments

% `myaudioc` *packet-size buffer-size target-buffer service-time pa-trigger na-trigger*

where *packet-size* is the data unit (as before), *buffer-size* is the maximum queue size, *target-buffer* is the target queue length Q^*, and *service-time* is the time interval (in msec) at which the buffer is dequeued/serviced, analogous to the router structure of Problem 3. Additionally, *pa-trigger* is an integer count by which a PA-type feedback packet is triggered (e.g., 10 would mean that every 10 contiguous packets a PA-type feedback is sent to the server) and *na-trigger* is an integer count by which NA-type packets are triggered: when a missing hole is detected (a jump in the sequence number), the current queue length is compared with *na-trigger*, and only if it is greater than *na-trigger* is a negative ACK sent to the sender (thus setting *na-trigger* to *buffer-size* would effectively disable NACK). The client, when dequeueing packets every *service-time* (msec) paced by SIGALRM, writes the data portion of the payload (the first 10 bytes contain book-keeping/control information) to the output device `/dev/audio`. It discards the packet, however, if the sequence number is outdated—that is, a packet with an earlier sequence number was already played. A headphone needs to be connected to a client PC in the Xinu Lab (in the back of the PC tower case, the audio jack below the microphone jack) to hear the output. If you don't have a headphone, please come by CS 220 and one will be provided. At the very start, the client waits until Q reaches Q^* before starting play—i.e., delayed playback—which means that the SIGALRM triggered service/playback must be postponed accordingly. Subsequently, playback is trigged by SIGALRM only, irrespective of queue size.

As in Problem 3, the client/receiver records the queue length—i.e., time stamp (in milliseconds from the start time) and queue length pair—when dequeueing a packet. Additionally, it records the sequence number upon dequeueing, thus forming a triple (time stamp, queue length, sequence number). Upon enqueueing a new packet, it records the sequence number (i.e., time stamp and sequence number pair). Write the log data into main memory and dump them onto disk at the end of the run to prevent unnecessary slow down. Thus two files should be generated. For each benchmark run, plot three graphs from the two log files: (1) the time series of the queue lengths, (2) the time series of the dequeueing sequence numbers (can only grow), and (3) the time series of the enqeueing sequence numbers (can decrease for retransmitted packets). These three plots, collectively, provide a performance description of each run. Lastly, for each run, record how many packets are ultimately missing at playback time (about 1000 packets will be generated for ~park/pub/cs536/pp.au without losses): 0 means perfect playback, and the higher the number the less effective NACK-based recovery has been. Show this number as a ratio (between 0 and 1), also called the miss rate.

Perform the following benchmarks: (1) *drop-prob* = 0.0 (no dropping), *rand-seed* = 7563; *buffer-size* = 40, *target-buffer* = 20, *service-time* = 100 (msec), *pa-trigger* = 20, *na-trigger* = 40; (2) same as above, except *drop-prob* = 0.1; (3) same as (2), except *na-trigger* = 10; (4) same as (3), except *na-trigger* = 1; (5) same as (4), except *target-buffer* = 10; (6) same as (5), except *target-buffer* = 3; (7) same as (6), except *target-buffer* = 1; (8) same as (4), except *drop-prob* = 0.2; (9) same as (4), except *drop-prob* = 0.4; (10) same as (4), except *drop-prob* = 0.7. Give an interpretation of your performance results.