CS 422 (Park) Assignment VI and Project Due: Apr. 24 (Tue.), 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

Read Chapters 25, 27, 31, 35, and 40 from Comer.

PROJECT PROBLEM

Design, implement and benchmark a UDP-based peer-to-peer (P2P) pseudo real-time audio streaming application. The sender transmits packets containing audio payload at rate λ. The rate may change over time when congestion control actions are undertaken upon receiving feedback control packets from the receiver. From a programming perspective, the application is an exercise in asynchronous signal handling (a continuation of Problem 3, Assignment V) where the sender's transmission of audio packets is paced by SIGALRM (e.g., by invoking `usleep`() between successive transmission of packets with sleep parameter $1/\lambda$). The receiver is more interesting in that it needs to invoke two signal handlers. One for SIGIO (or SIGPOLL) that is triggered when an audio packet from the sender arrives which is written into the application's audio buffer. Another for SIGALRM which is triggered periodically at a specified playback rate at which time an audio sample is taken out from the audio buffer and played back (i.e., written to `/dev/audio`). Thus at the receiver, SIGIO drives the producer side of the audio buffer whereas SIGALRM drives the consumer side. Of course, mutual exclusion must be assured so that the audio buffer data structured is not corrupted by concurrent access by producer/consumer.

The sender, `my_audio_send`, takes as command-line arguments

% `my_audio_send` *dest-IP dest-port audio-file payload-size packet-spacing mode*

where *audio-file* is a stored audio file that will be streamed to the receiver—unless otherwise indicated, assume the file format is binary—*payload-size* (in bytes) is the size of the UDP payload (excluding a 2-byte sequence number inscribed at the start of the payload) at which unit the audio file will be segmented and transported, *packet-spacing* (msec) is the initial packet spacing used in the transmission of the audio file, and *mode* specifies the congestion control mode: 0 (method A), 1 (method B), 2 (method C), and 3 (method D) discussed in class. The receiver, `my_audio_rcv`, has command-line arguments

% `my_audio_rcv` *port-number log-file payload-size pb-del pb-sp buf-sz target-buf*

where *pb-del* is the initial playback delay (sec)—time delay from the arrival of the first audio packet—*pb-sp* (msec) is the time interval at which buffered audio is written to `/dev/audio` for playback (triggered by SIGALRM), *buf-sz* is the total allocated buffer space (bytes), and *target-buf* (bytes) is the target buffer level (i.e., Q^*). In the receiver's code structure, attention needs to be paid to the shared audio buffer—the SIGIO handler will write to the buffer when audio packets arrives whereas the SIGALRM handler will read from the buffer for audio playback—so that it does not get corrupted due to concurrent access (e.g., semaphores may be used to achieve orderly access). When audio packets, upon arriving, find the audio buffer full, they will be dropped.

To monitor how well the system is performing (we are not entirely relying on the ears and the stuff that sits between them), the sender logs the current sending rate λ (= 1 / *packet-spacing*), along with the time stamp from `gettimeofday`(), whenever a packet is transmitted. The receiver, upon receiving an audio packet from the sender (SIGIO handler) or dequeueing an audio packet at playback (SIGALRM handler), logs the current time stamp and buffer occupancy $Q(t)$ for off-line diagnosis. Measurement logs should be written to main memory and flushed to disk at the end of the run to avoid overhead/slow-down stemming from disk I/O. To affect congestion control, the receiver transmits a feedback packet containing $Q(t)$, Q^* (i.e., *target-buf*), and γ (in terms of time interval *pb-sp*,

not rate) to the sender. Depending on the *mode* value, the sender will utilize the received information to institute the chosen congestion control.

Benchmark the application between two machines where a `.au` *audio-file* is provided (see TA notes), payload size is 256 B, initial *packet-spacing* at the sender side is 70 msec, *pb-del* is 4 seconds, *pb-sp* is 40 msec, *buf-sz* is 40 KB, and *target-buf* is 20 KB. Perform two benchmark runs per congestion control method. Plot the time series measurement logs using `gnuplot` at the sender, $\lambda(t)$ against time t, and packet rate (pps) as a function of time at granularity 1 sec; at the receiver, the queue length time series that plots $Q(t)$ against t, and the received packet rate (pps) at 1 sec time granularity. Discuss your results and findings. Compare the audio quality perception (you need to use a headphone) with the numeric performance findings. Note that there is some degree of freedom in the selection of the congestion control parameters. Determine parameter settings for each method that you find work well (relatively speaking). The top 5 performing applications, as determined by the TA based on actual audio sound and performance logs, will be given 30 bonus points. Before running the receiver, make sure to set the audio controls using `audioctl` (see TA notes for instructions on how to use the command) so that audio can be output through a headphone jack in the PC (you can use your MP3 player, walkman, or any other compatible head set). Do not use external speakers.

Use `ftp` to time (approximately) how long it takes to transfer the entire audio file using TCP and how the transmission profile (the rate at which TCP transfers the bits in the file changes over time) looks like. To do that, run `tcpdump` at the receiver. From the logs, determine how long it would have taken `ftp` to reach Q^*. The comparison is not entirely fair since the UDP-based P2P audio streaming application uses a payload size of 256 B. From the logs, plot the data rate (pps) at 1 sec granularity. Compare `ftp`'s data rate with that of the UDP sender's logged data rate. Discuss your findings.