CS 422 (Park) **Assignment VI** **Due: Dec. 9 (Fri.), 2011**

Submission instructions: Please type your answers and submit electronic copies using `turnin` *by 11pm 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 format that uses standard fonts (a practical test is to check if the pdf file prints on a CS Department printer). For experiments and programming assignments that involve output to terminal, please use* `script` *to record the output and submit the output file.*

PROBLEM 1

Read chapters 26–30, 32.

PROBLEM 2 (120 pts)

A variation of Problem 2, Assignment V, rewrite the server code in Java so that it can be run on the Linux PCs in our lab. Make a slight modification to the message format used by client and server so that it is compliant with HTTP/1.1. The client only needs to issue the GET method to request a file (see TA Notes for format related information) with minimal header line "Host" to specify the server name. Let DNS do some translating work by using symbolic names (e.g., `sslab01.cs.purdue.edu`) in place of dotted-decimal IP addresses in the command-line input of the client. Test the client by running it with arguments that correspond to the URL

`http://www.cs.purdue.edu/homes/park/exponential.html`

The returned file, when viewed as a local HTML file from a brower, should look the same as when accessed directly from the browser by specifying the URL. Test the client/server as in Problem 2, Assignment V, with one additional case to trigger the "400 Bad Request" response. The server needs to support only 3 response types: "200 OK" (with minimal header lines of your choice) that is followed by the body of the response containing the file content (for simplicity an ASCII file), "404 Not Found" indicating that the requested file was not found, and "400 Bad Request" indicating that the request could not be understood.

PROBLEM 3 (200 pts)

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 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 $1/\lambda$). Although our discussion will use rate λ as the control variable to be consistent with the material covered in class, when implementing the sender-side control it is easier to use the time spacing between packets (i.e., $1/\lambda$, call it τ) directly rather than updating λ and then computing $1/\lambda$. Hence to increase the packet sending rate your sender would decrease τ, and vice versa to slow things down. The receiver is more interesting in that it needs to invoke two signal handlers. One for SIGPOLL (or SIGIO) 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, SIGPOLL drives the producer side of the audio buffer whereas SIGALRM drives the consumer side. Use a binary semaphore so that mutual exclusion is assured, i.e., 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 4-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. Also, devise your own method, selected by mode 4 (method E). Include a half-page write-up that explains what your congestion control is aiming to do that is different from methods A–D.

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 SIGPOLL 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. When audio packets, upon arriving, find the audio buffer full, they are 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 (SIGPOLL 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 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 selected congestion control method.

Benchmark the application between two machines where a `.au` *audio-file* is provided (see TA notes), payload size is 250 B, initial *packet-spacing* at the sender side is 90 msec, *pb-del* is 3 seconds, *pb-sp* is 30 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. The top 3 performing method E congestion controls, 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 additional instructions) so that audio can be output through a headphone jack in the PC.