CS 422 (Park) Assignment VI and Project Due: Apr. 24 (Mon.), 2006

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 13, 16–21, 24–27, 31, 35, and 40 from Comer.

PROBLEM 2 (50 pts)

As a continuation of Problem 2, Assignment V, reimplement the client/server application so that it uses TCP sockets in place of UDP. Most of your code will remain the same, the changes relating to replacing the UDP based client/server interaction with a TCP socket interface. Test your client/server system on the same benchmark suite.

PROBLEM 3 (70 pts)

Also a continuation of Problem 2, Assignment V, reimplement the client/server application so that the UDP server is made into an *iterative server* that performs the requested task itself. (Note: Hereto our servers have been *concurrent servers* that only parse incoming requests, handing off the actual servicing of the task to child processes.) The function performed by the iterative UDP server is not remote command execution anymore—that is impossible to do without forking another process to `exec` the requested command—but instead performs UDP layer `ping` service. That is, when a suitably formatted UDP `ping` request packet is received (the default `ping` application uses a lower layer protocol called ICMP), the packet is "bounced back" to the sender (i.e., the source and destination IP addresses/port numbers are switched). The command to be executed on the client side is

% `my_ping` *IP-address port-number size count packet-spacing*

where *IP-address* and *port-number* are the coordinates of the UDP `ping` server, *size* (bytes) is the payload size to be used by `my_ping`, *count* is the number of packets to be sent by the client, and *packet-spacing* (msec) is the time interval between successive packets. The first 5 bytes of the payload is a simple password field so that only if "terve" is inscribed will the `ping` server respond. The next 4 bytes contain an unsigned integer that serves as a sequence number. The remainder of the packet is padded with *count* bytes of the ASCII character 'Z'. Implement *packet-spacing* using `ualarm`() (not `usleep`()) to pace the probe packets at constant packet rate of 1000 ÷ *packet-spacing* pps. Thus a signal handler for SIGALRM will perform the transmission of probe packets. `my_ping` numbers the probe packets 0, 1, ..., and just before transmitting a packet using `sendto`() (do not use `write`() with `connect`() in the iterative UDP client/server) calls `gettimeofday`() to record the send time for RTT calculation when the reflected packet returns. The time stamp is stored in an array of a `struct` containing two fields for sequence number and time stamp so that the response packets can be correctly matched to their probe packets when calculating the RTT. This job is performed by another signal handler on the client side for SIGIO (or SIGPOLL). The SIGIO handler prints the calculated RTT value (in microsecond unit), the sequence number, and the time stamp at the client when the response is received. The server code is simple, and is invoked by

% `ping_server` *port-number*

Benchmark the UDP `ping` application by running the server and client on different machines. Perform runs with parameters *size* 64 B, *count* 100, and *packet-spacing* 100 msec. Use `script` on the client to record the interaction and output. Use `gnuplot` with style 'impulse' to show which probes have successfully processed responses (i.e., the x-axis is the sequence number 0–99 and the y-axis shows an impulse when a response has been received). Plot a similar impulse plot but with the x-axis denoting the time stamp when a response is received. Normalize the time stamp value so that it starts from 0 and the unit is in microseconds. Repeat the experiment with *packet-spacing*

values 50, 10, and 1 msec (the other parameters remain the same). This yields 6 additional impulse plots. For each of the 4 runs, calculate how many responses were logged (maximum 100). Plot the results using `gnuplot` where the x-axis is *packet-spacing* and the y-axis is the number of logged responses. Do the same where the y-axis is the (approximate) completion time: time stamp of last response received minus time stamp of first response received. Carefully examine the plots and discuss the results. Questions to address include if performance changes when *packet-spacing* is decreased from 10 msec to 1 msec. If not, why is that the case? Your discussion should address two parts: the general trend you observe in the data and any anomalies (and an explanation of why they are likely to occur). Repeat the four runs where *size* is increased to 1 KB. Plot the response count and completion time plots. To contrast with the *size* 64 B runs, plot the 1 KB and 64 B results in the same plot. There is no need to plot the impulse plots. Discuss the results.

PROJECT PROBLEM

Design, implement and benchmark a UDP-based peer-to-peer (P2P) pseudo real-time audio streaming application. Your application can be built on top of the UDP `ping` application in Problem 3 with modifications to inject congestion control and audio playback components. 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 (i.e., client)—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 constant bit rate (CBR) 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). 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.

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 12 byte feedback packet containing Q^* (i.e., *target-buf*), $Q(t)$, 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 512 B, initial *packet-spacing* at the sender side is 80 msec, *pb-del* is 5 seconds, *pb-sp* is 40 msec, *buf-sz* is 70 KB, and *target-buf* is 40 KB. Perform one benchmark run 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 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). 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.

Repeat the benchmark for playback delay *pb-delay* 10 seconds and 1 second for `method D` only. Compare the results to the 5 second delay case. Based on your experimental findings, discuss what you would consider to be effective parameter settings for CD quality audio playback in today's Internet where end users have (at least) broadband connections.