CS 422 (Park) **Assignment VI** **Due: April 29 (Fri.), 2011**

Submission instructions: Please type your answers and submit electronic copies using `turnin` *by 4pm 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. Use* `gnuplot` *to plot graphs.*

PROBLEM 1

For students who have the 5th edition: Read chapters 27–32. For students who have the 4th edition: Read chapters 27, 31, 33, 35, 39–40.

PROBLEM 2 (50 + 50 + 80 pts)

(a) Identify the vulnerabilities of your `my_talk` messaging app of Assignment V, Problem 3(a), that may be exploited by a network-based attack. Network-based means that one or more packets are sent to the UDP port used by the messaging app with the aim of causing significant disruption to the application including disabling, crashing, or hijacking the app. After describing the vulnerabilities, implement attack programs that exploit the vulnerability and demonstrate that they work through tests that you set up. The tests must be repeatable (if meaningful provide scripts) so that the TAs can evaluate your attack exploits and their impact on `my_talk`.

(b) Come up with solutions to mitigate the vulnerabilities identified and tested in part (a). Discuss how well your solutions address the attack exploits. Implement the solutions by modifying your `my_talk` app. Using the tests of part (b), demonstrate how the attacks are mitigated: in the best case, the attack no longer works, in other cases the negative impact may reduced.

(c) Repeat parts (a) and (b) for the TCP-based remote command server code—vulnerabilities on the client side are ignored—of Problem 3, Assignment IV.

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 $1/\lambda$). 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 260 B, initial *packet-spacing* at the sender side is 100 msec, *pb-del* is 4 seconds, *pb-sp* is 40 msec, *buf-sz* is 30 KB, and *target-buf* is 15 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 D congestion controls, as determined by the TA based on actual audio sound and performance logs, will be given 30 bonus points. The top 3 method E congestion controls will be given 50 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.

Use `scp` 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 `scp` to reach Q^*. From the logs, plot the data rate (pps) at 1 sec granularity. Compare `scp`'s data rate with that of the UDP sender's logged data rate. Discuss your findings.

BONUS PROBLEM (150 pts)

The bonus problem provides extra points for the homework component of the course. It only counts if Problems 2 and 3 of Assignment VI are completed.

Design, implement, and test a UDP-based reliable file transfer protocol that uses negative ACK to achieve reliability above the UDP transport layer. The server (i.e., sender), call it `my_ftps`, waits for requests on a designated port, and given a client request (call the client `my_ftpc`) that specifies a full pathname or a file name that is searched in the current directory of the server process transmits the content of the file using a form of ARQ that uses negative acknowledgments. There is a lot of degree of freedom in the form of ARQ implemented, the only requirement being that it uses negative ACKs to trigger retransmissions when a packet is deemed lost by the server and that the protocol is correct. When testing, first demonstrate that the file transfer application works correctly. Second, compare the performance of your app against that of `scp` that uses TCP. Since `scp` incurs additional overhead due to encryption, your application has an advantage. Measure the file transfer completion time using the `time` command. For `scp`, subtract the approximate time taken to input your password by using a timer/watch. Focus on large files in the performance comparison which should yield more robust performance averages. If your application has parameters that impact its performance, explain how you chose them to optimize performance.