CS 536 Fall 2021

Lab 5: Congestion Control for Audio Streaming [250 pts]

Due: 11/15/2021 (Mon), 11:59 PM

Objective

The aim of this lab is to implement feedback congestion control for pseudo real-time multimedia streaming. To not get sidetracked by video/audio encoding standards and related issues, we will use audio streaming which allows us to focus on the networking components. The same methods apply to video streaming.

Reading

Read chapters 5 and 6 from Peterson & Davie.

Problem 1 (250 pts)

1.1 Audio playback on lab machines (both pod and amber)

Prior to CS Linux lab machines transitioning to Ubuntu Linux, our Linux distros supported /dev/audio that provided a simple interface to play back audio. For example, to play back an audio file in .au format calling open() on /dev/audio and writing the content of the .au file sufficed. On command line, % cat somefile.au > /dev/audio, sufficed for playback on the default audio device of a lab machine running Linux or UNIX. With cessation of /dev/audio support in Ubuntu on our lab machines, a slightly more complicated interface must be used to play back the content of an .au audio file on the default ALSA audio device of the pod and amber Linux PCs. To get familiar with performing audio playback, copy two sample audio files pp.au (short piece) and kj.au (longer piece) from the course directory, and run

```
% /usr/bin/aplay pp.au
```

with an earphone plugged into a pod or amber machine (audio jack is located on the left edge of the display). If you don't hear anything, check that audio is not muted (under Preferences -> hardware -> Displays -> Sound configuration tabs in Ubuntu MATE). Do the same for kj.au which is a bit longer. If you don't have access to an earphone, please contact the TAs or me.

After playback is confirmed using aplay, you will need to code playback within the client code of the pseudo real-time audio streaming app. To do so, inspect the code testaudio.c which is available in the course directory. Similar to aplay, testaudio.c takes an audio filename as command-line argument and plays the content of the file on the ALSA audio device. A second command-line argument specifies the time interval (in microseconds) between successive writes to the audio device. Set this value to 313 msec (i.e., 313000). Test that it works by compiling testaudio.c with library -lasound and running it with the two .au audio files.

To port the playback component to your audio streaming client, copy the code pertaining to mulawopen(), mulawclose(), mulawwrite() including header files and variables to your client app. The code in testaudio.c reads the content of an .au file in unit of 4096 bytes (bufsiz) into main memory (buf), then writes the 4096 bytes contained in buf to the audio device by calling mulawwrite(). Between successive calls to mulawwrite(), testaudio.c calls usleep() to sleep for a fixed period given by the argument slptime. The parameter slptime (in unit of microseconds) is provided as the second command-line argument of testaudio.c. Relating slptime to the playback rate gamma in the pseudo real-time streaming model, the larger slptime the smaller gamma since the latter is a rate. The recommended value of slptime to use in your client is 313 msec (i.e., 313000 microseconds). Use nanosleep() instead of usleep() to sleep between successive read/write of 4096 bytes to the audio device in your client code.

1.2 Pseudo real-time audio streaming

Implement and evaluate an UDP-based pseudo real-time audio streaming application following the congestion control framework for multimedia streaming discussed in class. The client, audiocli, sends an UDP request packet to the server containing the name of the file to be streamed. It also sends a secret key and blocksize following the same format as Problem 1 of lab4 (which is the same format as Problem 2 of lab3). The server, audiosrv, performs sanity checks as before with the added constraint that the filename must end with postfix ".au". If the request is valid, the parent process forks a child process that is then tasked to perform the actual streaming of the requested .au audio file.

The child process creates a new UDP socket with the same IP address but a different unused/ephemeral port number. It uses the new socket to perform subsequent communication with the client using sendto()/recvfrom() system calls. The parent process goes back to waiting for new client requests.

1.3 Server structure

The server is invoked with command-line arguments

```
% audiosrv srv-IP srv-port lambda logfilesrv
```

where srv-IP and srv-port specify the server's IP address and well-known port number that the parent binds to. lambda is the initial sending rate in unit of pps (packet per second). The last argument logfilesrv specifies the name of a log file where the changing sending rates are recorded for diagnosis at the end of a streaming session. Since the server can handle multiple clients, logfilesrv of the first client should be appended with "1", logfilesrv1, the second client's log file with "2", logfilesrv2, etc., to distinguish their log files. The values of changing lambda values must be saved in an array in main memory during the streaming session. Only at the end of the streaming session before a child process terminates is the array written to logfilesrv. This avoids introducing side effects stemming from disk I/O overhead. Along with changing lambda values, a timestamp is saved for each logged lambda value. Use gettimeofday() to get timestamps but subtract the first timestamp from successive timestamps so that time starts at 0 for the first logged lambda value and successive lambda values are relative to the first lambda value.

Although lambda in the command-line argument specifies a rate in unit of pps, for actual transmission and control it is simpler to use 1/lambda, i.e., time interval between successive packet transmission as the control variable to adjust. We will call 1/lambda, pspacing, maintained in unit of millisecond. Use nanosleep() with pspacing as input to affect how fast the server sends audio data to the client. When logging lambda, record pspacing which will be easier to interpret when diagnosing performance at the end of a streaming session.

For our audio streaming app, we will adopt a dumb sender/smart receiver design where a feedback packet is sent from client to server (i.e., child) that specifies the pspacing value to use next. An UDP feedback packet received from a client contains as payload two bytes which represents a value of type unsigned short that overwrites the previous pspacing value. The unsigned value is in unit of msec that either increases, decreases, or keeps pspacing unchanged. pspace is logged whenever a data packet is transmitted to the client, not when a feedback packet from the client is received. The command-line lambda value is only used to calculate the initial pspacing value 1/lambda to be used before the first feedback packet from the client arrives. Subsequently, the payload in feedback packets specifies updated pspacing values to use for subsequent audio data transmissions. The size of UDP's payload when sending a data packet to the client is determined by the blocksize value when a streaming session request is received from the client. After the last data packet has been transmitted, audiosrv transmits 8 UDP packets containing no payload signifying that the session has ended. The child process writes the logged values to logfilesrv and terminates.

1.4 Client structure

The client, audiocli, runs with command-line arguments

```
% audiocli srv-ip srv-port audiofile blocksize buffersize targetbuf lambda method logfilecli
```

where srv-ip is the server's IP address, srv-port the server's port number, audiofile the name of the .au file (including ".au" extension/postfix and limited to 8 bytes total) to be streamed, blocksize retains the same role as in lab3 and lab4. buffersize specifies the size of the client's FIFO buffer where incoming audio data are stored before playback on the ALSA audio device. The unit of buffersize is in multiples of 4096 bytes which is the fixed size that mulawopen() and mulawwrite() are configured with. If blocksize is smaller than 4096, it may happen that the client's FIFO buffer is not empty but contains less than 4096 bytes. In such cases, the client process does not read the bytes in its buffer to send to the audio device by calling mulawwrite() but treats it as if empty. Buffer occupancy must be greater or equal to 4096 bytes for playback to occur. The parameter targetbuf, also in multiples of 4096 bytes, is the target buffer occupancy Q* we are aiming to achieve. lambda is the same initial lambda value used by the server. The argument method specifies which congestion control method to use: 0 for method C and 1 for method D. The last argument logfilecli specifies a log file where the client logs the changes to its buffer occupancy.

The client's FIFO buffer is a classical producer/consumer queue where the producer is the code that calls recvfrom() to receive the next data packet from the server and copies its payload to the FIFO buffer. The consumer is the code of the client that every 313 msec (use this fixed value which yields acceptable audio quality when the buffer is never depleted) reads 4096 bytes from the buffer and calls mulawwrite() to send the audio content to the audio device. You must use standard concurrency control methods such as semaphores to prevent corruption of the shared FIFO data structure. For example, in one implementation recvfrom() may be implemented synchronously that blocks on arrival of data packets from the server, whereas reading from the FIFO buffer is implemented asynchronously by coding a signal handler that is invoked every 313 msec. Or vice versa, and other implementations using select(). Current size of the FIFO buffer (i.e., Q(t)) is logged into an array in main memory whenever there is a write or read operation performed on the buffer. Buffer size is recorded along with timestamps obtained from gettimeofday(), but normalized to start from 0 in unit of milliseconds as in the server audiosrv. Just before the server terminates, the logged values are written to logfilecli.

When the client starts up, it sends a streaming request packet containing a 2-byte blocksize bytes followed by the filename (up to 8 bytes). We will not utilize a secret key in this problem. The 2-byte block size can encode a value up to 4096, with smaller sizes such as 1472 (bytes) fitting inside an Ethernet frame (1472 = 1500 - 20 - 8). It sets a timer as in Problem 1, lab4, to resend the request if a data packet from the server does not arrive. Note that the server's IP address will be the same but its port number will have changed since the data packet is being sent by a child process. In our dumb sender/smart receiver design, it is the client who, based on current buffer occupancy (i.e., Q(t)), targetbuf, and current lambda (1/pspacing) for congestion control method C, and additionally gamma (1/313 msec) for method D computes an updated lambda and resultant pspacing value 1/lambda which is communicated via a feedback packet to the server. This is performed whenever a change to the FIFO buffer occurs. For method C the epsilon parameter (i.e., gain constant) is needed, and for method additional gain parameter beta. Read these values from a file, audiocliparam.dat, when the client starts.

1.5 Performance evaluation

To evaluate how well the pseudo real-time streaming app performs, plot the client and server log files which will show if the system is converging to the target buffer occupancy targetbuf, and, if so, how fast. Run the client on a pod machine with the server on either pod or amber machine. Configure buffersize in the mid 50 KB (multiple of 4096 bytes) range, and targetbuf half its size as a reasonable starting point. Experiment with different blocksize, initial lambda values, and gain parameters. Evaluate methods C and D for the same configuration. Plot the the time series measurement logs using gnuplot, MatLab, or Mathematica which will help diagnose how your streaming control is performing. gnuplot is very easy to use and produces professional quality graphical output in various formats for inclusion in documents. Discuss your findings in lab5.pdf. Submit your work in v1/ following the convention of previous labs (e.g., Makefile, README).

Note: This problem may be tackled as a group effort involving up to 3 people. If going the group effort route, please specify in lab5.pdf on its cover page who the members are, who did what work including *programming the various client/server components, performance evaluation, and write-up. Whether you implement lab5 as an individual effort or make it a group effort is up to you. Keep in mind the trade-offs: group effort incurs coordination overhead which can slow down execution, especially for a 2-week assignment. Benefits include collaborative problem solving and some parallel speed-up if efficiently executed. Regarding late days, for a group to use k (= 1, 2, 3) late days, every member of the group must have k late days left.*

Bonus Problem (40 pts)

Add a new control law, method E, selected by value 2 of the method parameter in the client's command-line arguments. The aim is improve upon the performance of method D. Describe your control method and its rationale in lab5.pdf. Try to be creative which starts with clearly stating the motivation. Even though an idea may seem reasonable, whether it actually pans out is another matter. Implement the method and evaluate its performance by comparing to the results of Problem 1. Submit the extended code in v2/. If Problem 1 is tackled as a group effort, the bonus problem must be solved as a group effort as well.

Turn-in Instructions

Electronic turn-in instructions:

We will use turnin to manage lab assignment submissions. Go to the parent directory of the directory lab5/ where you deposited the submissions and type the command

```
turnin -v -c cs536 -p lab5 lab5
```

Back to the CS 536 web page