EECS 489 Lab 7: Rate Control with Token Bucket Filter

This assignment is due on Wednesday, 8 April 2015, 10 pm.

Introduction

We have seen how sliding window flow control allows the receiver to throttle sender's transmission so as not to overflow the receiver's buffer. By relying on returning ACKs to slide the sender's window forward, sliding window flow control is a close-loop system. In this lab we investigate the use of open-loop sender rate control. The advantage of open-loop control is obviously that we don't have to wait one round-trip time for ACK packets to drive sender transmission. Instead, the receiver specifies a token-bucket filter that governs how bursty and how fast on average the sender can transmit. Since rate control by itself does not provide reliability, packets can still be dropped or arrive out of order.

We start with a UDP-based client-server code with no flow control nor reliability, very similar to the client in Lab5. The only difference is that the client takes an additional command line argument, `-r`, which is used to specify the flow rate, in Kbps. (The reason the flow rate is specified at the client side instead of the sender side will become apparent in Lab8.) You are provided with the full client source code, so no client reference binary is provided. A reference Linux binary executable of the server `refimgdb` is available in the Course Folder. A tarball of the support code is also available in the Course Folder. The provided `Makefile` builds both `netimg` and `imgdb`. The two tasks you're asked to do in this lab both reside on the server. Your implementation of the server **must** interoperate with the provided client code. Since the provided client code reveals partial solution to Lab5 and PA3, you will be granted access only after you're done with your PA3 or have decided not to complete it. As usual, you can search for the string "YOUR CODE HERE" in the code to find places where your code must go. You may also want to consult the PA4 walk-through lecture notes.

Task 1: Token-Bucket Filter Sizing

The `imgdb` server in this lab does not take any command line option. The image file the client queries must reside in the same folder from which the server is launched. In `imgdb::handleqry()`, the flow rate specified in the client's query packet is stored in `imgdb::frate` for you.

Your first task is to compute, in `imgdb::handleqry()` the token bucket size and token generation rate to regulate the sender's transmission. The token bucket size should be large enough to hold at least as many tokens as are needed to cover one `mss` segment, excluding all headers. If the receiver's window, `rwnd`, is larger than 1 `mss`, the token bucket size should be sized such that the server can fill the receiver's window, but no larger. You may find the macro `IMGDB_BPTOK`, defined in `imgdb.h`, useful in doing this computation. This macro specifies the number of bytes corresponding to a single token. For example, if your `mss` is 1460 bytes and `IMGDB_BPTOK` is 512 bytes, you'll need to have 3 tokens in your token bucket to send a single segment. To ensure transmission progress, your token bucket must be sized large enough to hold at least the number of tokens neeeded to send a single segment. Next compute the token generation rate, which is simply a translation of the user-specified sender's rate from Kbps to tokens/sec. Store the computed token-bucket size in `imgdb::bsize` and the token rate in `imgdb::trate`.

You can search for the string "Task 1" in `imgdb.cpp` to find the one place to enter your 2 lines of code for

sizing the token-bucket filter. And that's all you have to do for Task 1.

Task 2: Rate-controlled Transmission

Your second task is to regulate the server's transmission rate. You can search for the string "Task 2" in `imgdb.cpp` to find the two places in `imgdb::sendimg()` where "Task 2" related code must be filled in. In `imgdb::sendimage()`, first you need to decide what local variables you need to regulate the transmission using the token bucket filter. Initialize your local variables. We will regulate only the transmission of `NETIMG_DATA` packets. For a segment to be transmitted, there must be enough tokens in the token bucket filter to cover the data portion of the packet, i.e., `mss` minus all headers.

To send a segment, you first check if there's enough tokens in your token-bucket filter to cover the segment. If there isn't enough tokens, put the process to sleep. The amount of time the process sleeps should at minimum be long enough to generate at least the number of tokens needed to cover a single segment. Preferrably, it should sleep a little longer than the minimum, to allow for more tokens to accumulate. For example, you could sleep for an additional amount of time sufficient to cover a random fraction of the token bucket size. Try to use a system call that allows for microseconds granularity in specifying the sleep time, e.g., on Unix system, use `usleep()` instead of `sleep()`. In this assignment, we assume that the time to transmit a bucket-full of data is negligible, compared to the time needed to generate a token, given the token generation rate. If the transmission time of a bucket-full of data is not negligible, we'd need to account for the tokens generated during that time also (which we won't do). In any case, be sure to enforce that your token bucket size is not larger than `imgdb::bsize`. Upon transmission of a segment, deduct the token(s) used from the amount of accumulated tokens. Token consumption can be fractional. For example, instead of using 3 tokens to send a segment of size 1460 bytes, you can use 2.85 tokens. The same applies to token generation: you can generate fractional tokens.

That's is for Task 2. It should take about 10 lines of code.

Testing Your Code

Given the appropriate receiver window size (`-w`) and sender's flow rate (`-r`), you should be able to specify a token bucket filter that allows you to transfer an image file without overflowing the receiver buffer and therefore can be displayed with no gap in the image, modulo dropped packets. A larger receiver window means that the sender can send a large burst at once. So unlike in previous assignments, a larger receiver window does not necessary result in reliable transfer with no dropped packets. A smaller sender's rate not only slows down transmission, it also slows down the token (re)generation rate. For example, on localhost, with the default receiver window of 10 packets and `mss` of 3KB:

```
% netimg -s localhost:<port> -q ShipatSea.tga -r 256
```

should take twice as long to complete as

```
% netimg -s localhost:<port> -q ShipatSea.tga -r 512
```

and four times as long as

```
% netimg -s localhost:<port> -q ShipatSea.tga -w 40 -r 1024
```

When connecting to CAEN from home over ADSL or cable modem, I found that

```
% netimg -w 10 -r 10 -m 10248
```

works for most images, just be prepared to wait 5 to 20 minutes for the image to be displayed and your mileage may vary.

Submission Instructions

Do NOT use any libraries or compiler options not already used in the provided `Makefile`, to ensure that we will be able to build your code for grading. If we can't compile your code, you will be heavily penalized.

Test your compilation on CAEN eecs489 hosts! Your submission must compile and run **without** errors on CAEN eecs489 hosts using the provided `Makefile`, unmodified. Your server must interoperate with the provided client code.

Your "*Lab7 files*" should comprise only your `imgdb.cpp`.

To turn in your Lab7:

1. Submit the SHA1's of your *Lab7 files* on the CTools Assignments page. (If the URL doesn't work for you, just click the "Assignments" item on the left menu of the CTools page for EECS 489.) Once you've submitted your SHA1's, don't make any more changes to the files, or your SHA1's will become invalid.
2. Upload your *Lab7 files* by pointing your web browser to Course folder and navigate to your `lab7` folder under your uniqname. Or you can `scp` the files to your `lab7` folder on IFS: `/afs/umich.edu/class/eecs487/w15/FOLDERS/<uniqname>/lab7/`.
   This path is accessible from any machine you've logged into using your ITCS (`umich.edu`) password. Please report any problems to ITCS.
3. Keep your own backup copy! Don't make any more changes to the files once you've submitted your final SHA1's.

The timestamp on your SHA1 submission on CTools' Assignments page will be your time of submission. If this is past the deadline, your submission will be considered late. You are allowed multiple "submissions" without late-policy implications as long as you respect the deadline.

Do NOT turn in an archival (`.zip` or `.tgz`) file, instead please turn in your solution files individually. Turn in ONLY the files you have modified. Do not turn in support code we provided that you haven't modified. Do not turn in any binary files (object files, executables, or images) with your assignment.

Do remove all `printf()`'s or `cout`'s and `cerr`'s and any other logging statements you've added for debugging purposes. You should debug using a debugger, not with `printf()`'s. If we can't understand the output of your code, you will get zero point.