EECS 489 PA4: Link-Rate Virtualization

This assignment is due on Monday, 20 April 2015, 10 pm.

Overview

In this programming assignment, you are to implement link-rate virtualization to support two clients. The first client schedules packets onto its virtual link using weighted-fair queueing (WFQ). The second client uses first-in first-out (FIFO) scheduling with token-bucket filter (TBF) to control its transmission rate and burstiness. The assignment consists of the following graded tasks.

Graded tasks (100 points total)

1. Link-rate virtualization (60 pts)
2. Client 1: Weighted-fair queueing (25 pts)
3. Client 2: FIFO with rate control using token-bucket filter (15 pts)
4. Writeup

The specification of this assignment relies heavily on Labs 7 and 8. The support code for this assigment consists of the support codes for Labs 7 and 8.

Assumptions

No client management

Ideally, our link virtualization mechanism can support a variable number of clients. For this assignment, we bake into the code support for only two clients. One way to implement this is by refactoring the `imgdb` class in labs 7 and 8 into three classes: a `FIFO` class, a `WFQ` class, and an overall `imgdb` class. Support for the two clients then simply requires an instantiation of the `WFQ` class and an instantiation of the `FIFO` class within the `imgdb` class. The provided `refimgdb` takes a new command line argument `-f` which specifies the fraction of link bandwidth to assign to client 1. Client 2 is then assigned the remainder of the link bandwidth.

No flow-setup protocol

Ideally, we have a flow-setup protocol to communicate to the link virtualization mechanism the parameters of each flow, including to which client a flow belongs. Instead, we assume that the client running the `FIFO` scheduler can support only one flow. The provided `refnetimg` has been modified to allow for flow setup with rate 0 (`-r 0`). We use this hack to set up a client 2 flow. When the provided `refimgdb` server receives an `iqry_t` packet with `iq_frate` set to 0, it automatically rewrites `iq_frate` to be the virtual link rate of client 2 and assigns the flow to client 2's FIFO scheduler. Any `iqry_t` packet with non-zero `iq_frate` is assigned to client 1 and is added to the client's WFQ schedule, space permitting.

No idle flow

As in labs 7 and 8, we assume no flow ever goes idle; that is, each flow always has something to send

until the end of transmission.

Link-Rate Virtualization

Command line options

On the client side, modify `-r` to accept 0 as a valid argument. On the server side, modify `-g` to accept 1 as a valid argument and add `-f` to specify client 1's fraction of the link rate.

Flow setup and transmission

In `imgdb::handleqry()`, if `iq_frate` of the incoming query packet is 0, set it to client 2's fraction of the link rate and pass the query along to client 2's FIFO query handler. Otherwise, pass it along to client 1's WFQ query handler. The FIFO query handler should check whether there's already an active flow in the FIFO queue. If so, return an `imsg_t` packet with `im_type` set to `NETIMG_EFULL` to the client. Otherwise, initialize the flow by calling `Flow::init()` to load the image and setup the necessary token-bucket filter variables as in Lab 7. The WFQ query handler can pretty much re-use the flow addition code in Lab 8. Back in `imgdb::handleqry()`, if a new flow has been added, update the `imgdb::nflow` member variable and start transmissions if the requisite number of flows have been added. (Since client 2's flow doesn't reserve a rate, we can no longer start transmission based on reserved rates reaching link rate.)

Once all the flows are added, `imgdb::sendpkt()` is called to transmit the next packet. When packets are transmitted back-to-back, the gap between packet *i* and packet *i*+1 is the length of packet *i* divided by the transmission rate. Since we assume all flows are active until they cease transmission, we simply alternate between the flows, sending the packet with the smallest transmission time first. One way to implement this is to write a `FIFO::nextxmission()` and a `WFQ::nextxmission` method that return the amount of time (in microseconds) until the transmission of the next packet from the FIFO and WFQ respectively. `imgdb::sendpkt()` then sleeps for the smaller of the two amounts of time by calling `usleep()`. Upon waking up, it sends the packet with the smallest transmission time, by calling `WFQ::sendpkt()` or `FIFO::sendpkt()` as appropriate, update the transmission time of the other client's packet by subtracting the amount of time spent sleeping, and repeat the whole process. You may want to have a member variable in the `imgdb` class to help you track the times until next transmission of both clients. You may want to initialize this variable and start the process of establishing the next packet to be transmitted in the `imgdb::handleqry` function. This is arguably the only "new code" you need to write for PA4. It should be less than 30 lines total.

The function `WFQ::nextxmission()` comprises the first half of `imgdb::sendpkt()` from Lab 8 that calls `Flow::nextFi()` and determines the smallest finish time. Once the packet with the smallest finish time has been selected, compute the transmission time of the packet, in microseconds, given the virtual link rate (fraction of link rate) given to the WFQ and return it to the caller. `FIFO::nextxmission()` comprises the Task 2 portion of `imgdb::sendimg()` from Lab 7. It may be convenient to call `Flow::nextFi` from `FIFO::nextxmission()` to compute the current segment size even if we don't need to compute any finish time. Similar to `WFQ::nextxmission()` compute and return the transmission time, in microseconds, of the next packet. If the token bucket is empty and we need to accumulate tokens, include this accumulation time, in microseconds also, in the return value. In this assignment, we assume tokens are accumulated only when the bucket is empty and during the computed token accumulation time, we don't accumulate token during transmission time. Finally, both `WFQ::sendpkt()` and `FIFO::sendpkt()` are variations on the second half of `imgdb::sendpkt()` from Lab 8 that calls `Flow::sendpkt()` and handles the accounting and cleanng up once a flow has completed transmission.

You may also want to consult the lecture notes on PA4 walk-through. Your implementation of the server **must** interoperate with the reference implementation of the client.

Testing Your Code

You may want to start by testing your WFQ and FIFO queues separately. Run `imgdb` with `-g 1` and connect a client to it, with `-r 0 to test the FIFO queue and -r non-zero to test the WFQ.`

Running the server with -g 2 should allow you to test 2 clients, each getting half of the link rate:

```
% netimg -q BlueMarble2004-08.tga -m 10240 -s localhost:<port>
% netimg -q BlueMarble2004-08.tga -m 10240 -r 0 -s localhost:<port>
```

Both flows should complete about the same time. However, specifying a smaller token bucket for the FIFO flow would let the WFQ flow to complete transmission first:

```
% netimg -q BlueMarble2004-08.tga -m 10240 -s localhost:<port>
% netimg -q BlueMarble2004-08.tga -m 10240 -r 0 -w 10 -s localhost:<port>
```

If you run the server with -g 3 -f .67, you can run three clients:

```
% netimg -q BlueMarble2004-08.tga -m 10240 -r 3430 -s localhost:<port>
% netimg -q BlueMarble2004-08.tga -m 10240 -r 3430 -s localhost:<port>
% netimg -q BlueMarble2004-08.tga -m 10240 -r 0 -s localhost:<port>
```

and they should all end about the same time. If you run the server with -g 4 -f .67 and four clients:

```
% netimg -q BlueMarble2004-08.tga -m 10240 -r 2286 -s localhost:<port>
% netimg -q BlueMarble2004-08.tga -m 10240 -r 2286 -s localhost:<port>
% netimg -q BlueMarble2004-08.tga -m 10240 -r 2286 -s localhost:<port>
% netimg -q BlueMarble2004-08.tga -m 10240 -r 0 -s localhost:<port>
```

the FIFO client should finish first, follow by the other three about the same time. These last two scenarios show that the FIFO flow received its allocated share of the link bandwidth and we have successfully virtualized the link rate across two clients.

And that's all the programming assignments for EECS 489. Thank you for taking the course!

Submission Instructions

As with PA1, to incorporate publicly available code in your solution is considered cheating in this course. To pass off the implementation of an algorithm as that of another is also considered cheating. If you can not implement a required algorithm, you **must** inform the teaching staff when turning in your assignment, e.g., by documenting it in your writeup. As with PA1, if you have not been able to complete Lab 7 and would like the solution so that you can complete this assignment, you may choose to forfeit the 15 points associated with it and obtain a solution from us. Similarly for Lab 8, you can get the solution by forfeiting the 25 points associated with it.

Test your compilation and build on CAEN eecs489 hosts! Your submission must compile and run **without** errors on CAEN eecs489 hosts. Code that does not compile on CAEN eecs489 hosts will be heavily penalized. Your solution must either work with the provided Makefile or you must provide a Makefile that works on CAEN eecs489 hosts. Don't 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. Your code **must** interoperate with the provided client code.

Create a writeup in **text format** that discusses:

1. Your platform and its version - Linux, Mac OS X, or Windows.
2. Anything about your implementation that is noteworthy.
3. Feedback on the assignment.
4. Name the file writeup-*uniqname*.txt.
   For example, the person with uniqname *tarukmakto* would create *writeup-tarukmakto.txt*.

Your "*PA3 files*" then consists of your writeup-uniqname.txt and your source codes.

To turn in your PA4:

1. Submit the SHA1's of your *PA4 files* on the CTools Assignments page. Once you've sent in your SHA1's, don't make any more changes to the files, or your SHA1's will become invalid.
2. Upload your *PA4 files* by pointing your web browser to Course folder and navigate to your pa4 folder under your uniqname. Or you can scp the files to your pa4 folder on IFS: /afs/umich.edu/class/eecs487/w15/FOLDERS/<uniqname>/pa4/.
   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 of 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. CTools keeps only your last submission.

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.

General

The General Advice section from PA1 applies. Please review it if you haven't read it or would like to refresh your memory.