EECS 489 Lab 6: Forward Error Correction

This assignment is due on Wednesday, 25 Mar 2015, 10 pm.

Introduction

This lab have you implement a simple Forward Error Correction (FEC) algorithm on top of the best-effort `netimg` application from Lab 5. Still assuming the absence of flow control and retransmission, we use a simple XOR-based FEC (or, a linear network code over GF(2)) to correct the loss of a *single* packet within a given window of data. In addition to the support files for Lab 5, you're also provided with `fec.cpp` and an updated `Makefile`. The reference Linux binary executable of the client for this lab is called `reffecimg`, and the server `reffecdb` (the provided `Makefile` still build `netimg` and `imgdb` though). You should be able to run your client against the provided server and the provided client against your server. There are no changes to the command line options of both apps from Lab 5. You can download the support code from the Course Folder and copy the file to your working folder for Lab5 (you may want to save a copy of your Lab5 `Makefile` before overwriting it with the one from this lab). As usual, you can search for the string "YOUR CODE HERE" in the code to find places where your code must go.

Task 0: Best-Effort Netimg

You'll need a working Lab 5 solution to start work on this lab. You can also obtain the solution to Lab 5 from the teaching staff at the cost of 15 points (ouf of 100) of your PA3 grades.

Assumptions

We make the following assumptions:

1. You have merged your Lab 5 solution with the provided support code.
2. We must work with the image buffer as returned by `LTGA::GetPixels()` without copying (including resizing) the buffer. Thus it is possible that the image buffer does not divide evenly into full maximum-segment-size (`mss`) packets.
3. When transmitting an image, all data packets except the last one are of maximum segment size (`mss`). The image data carried by each packet (the `datasize` henceforth) is mss minus the size of all headers, including the UDP and IP headers (`mss-sizeof(ihdr_t)-NETIMG_UDPIP`).
4. As in Lab 5, each byte is assigned a sequence number and the sequence number of a packet is the sequence number of the first data byte carried by the packet.
5. One FEC packet is sent for every `NETIMG_FECWIN` or receiver's window `rwnd`-1 number of segments, whichever is smaller. The last FEC window carrying the remaining image data may be smaller than the preceding FEC windows.
6. Each FEC packet carries an `ihdr_t` header and a data portion of size `datasize` (FEC data henceforth). When a data packet carries less than `datasize` amount of image data, the FEC data is updated *as if* the image data has been padded with 0s up to `datasize`.
7. FEC is computed over the image data of each packet only, not including the headers.

Task 1: Server Side

Your first task is to update the server code. You can search for the string "Lab6 Task 1" in `imgdb.cpp` and `fec.cpp` to find places where "Task 1" related code must be filled in. When client sends a query to the server, it also specifies the size of the FEC window (`iq_fwnd`) to use. In `imgdb_recvqry()`, the client-specified FEC window size is stored in the `imgdb::fwnd` member variable. The FEC window size is the number of segments from which each FEC data is computed. Think of the FEC window as advancing over the image data, jumping one window-full at a time. Each time, prior to an FEC window advance, you must generate an FEC data computed over all the segments within the current FEC window. FEC data is computed over the image data carried in each packet, not including the packet headers, not even the `ihdr_t` carrying the sequence number of each packet. The format of the query packet is the same as in Lab 5 and is defined in `netimg.h`. See the first three slides of the PA3 walk-through lecture notes for an illustration.

All the changes to the server code are in the function `imgdb_sendimg()`. You need to figure out how to:

1. maintain your FEC window,
2. keep track of your progress over each FEC window, and
3. compute your FEC data across multiple data segments.

Prior to transmitting a data segment, if this is the first segment in an FEC window, use it to initialize your FEC data. Subsequent segments within the FEC window should be XOR-ed with the content of your FEC data. Remember that FEC data is computed over the image data only, not over any headers, not even the `ihdr_t` header. The last segment in an FEC window may be smaller than `datasize`. Update your FEC data by XOR-ing it with the content of this last segment, then XOR the remainder of the FEC data beyond the size of the last segment with 0's up to `datasize`. You MUST use the two helper functions `fec.cpp:fec_init()` and `fec.cpp:fec_accum()` to encapsulate your FEC computation for the first and subsequent segments of the FEC window. You are to write these two functions. If you reach the end of the image before filling an FEC window, just send out the FEC data accumulated so far. The receiver will be smart enough to know that the FEC data has been computed over less than an FEC-window full of data. This task should take about 6 lines of code. The two helper functions each takes about 5 lines of code

If one FEC-window full of data has been transmitted or if the last segment of the image data has been transmitted, send out the FEC data. You can re-use the `struct msghdr` used for transmitting the last data segment by pointing the second entry of its `iovec` to your FEC data, which must be of size `datasize`. The header of the packet carrying FEC data MUST have `ih_type` set to `NETIMG_FEC`. The sequence number (`ih_seqn`) of an FEC packet MUST be the sequence number of the first image data byte *beyond* the FEC window. Don't forget to convert all header fields of type integer to network byte order. Call `sendmsg()` to send your FEC packet. This task takes about 10 lines of code.

That's all for Task 1. It should take less than 30 lines of code in total.

Task 2: Client Side

Your second task is to update the client code. You can search for the string "Lab6 Task 2" in `netimg.cpp` to find places where "Task 2" related code must be filled in. The provided function `netimage_sendqry()` compute the FEC window size `fwnd` and send it to the server in the `iq_fwnd` field of the `iqry_t` packet.

The function `netimg_recvimg()` is where all your work for this task is to be done. We will be re-using the `struct msghdr` to receive both image data and FEC packets. As with the server side, you need to figure out how to:

1. maintain your FEC window,
2. keep track of your progress over each FEC window, and
3. compute your FEC data across the multiple data segments.

To reconstruct a lost packet, one would normally store an FEC-window full of data (minus the lost packet) and wait for the FEC data packet to compute the lost packet. In our case, we don't "buffer" the image data, instead, as soon as a data packet arrives, we store it directly in the image data buffer. So you only need to keep track of which sequence number marks the start of the current FEC window. Our simple FEC scheme can only reconstruct one lost packet per FEC window. You only need to remember the location/offset of the first lost packet within the image buffer (out-of-order packet is considered lost). However, to avoid the futility of reconstructing the first lost packet when multiple packets per FEC window have been lost, you should also keep a count of total number of packets received per FEC window.

Upon receipt of each data packet, check whether it is the next data packet you're expecting. If not, you've lost a packet, mark the location of the first lost packet in an FEC window. Otherwise, increment your count of total number of packets received within the current FEC window. This task takes 3 lines of code.

You can re-use the `struct msghdr` to receive both image data packet and FEC data packet. The 5-line of code to do this is an adaptation of your Lab 5 code. Upon receipt of an FEC data packet, check if you've lost only one packet within the FEC window, if so, reconstruct the lost packet. Remember that we're using the image data buffer itself as our FEC window buffer and that you've noted above the sequence number that marks the start of the current FEC window. To reconstruct the lost packet, use `fec.cpp:fec_accum()` to XOR the received FEC data against the image data buffered, starting from the start of the current FEC window, one `datasize` at a time, skipping over the lost segment, until you've reached the end of the FEC window. If `fec_accum()` has been coded correctly, it should handle the case when the last segment of the FEC-window is smaller than `datasize`.

Once you've reconstructed the lost segment, copy it from the FEC data buffer to the correct offset on the image buffer. If the lost segment is the last segment of the image data, it may be smaller than `datasize`, in which case, you should copy only the correct amount of bytes from the FEC data buffer to the image data buffer. If no packet was lost in the current FEC window, or if more than one packets were lost, there's nothing further to do with the current FEC window, just move on to the next one. This task takes about 11 lines of code.

This task is further complicated by the following cases: (1) the FEC data packet itself may be lost, (2) multiple packets within an FEC window may be lost, and (3) one or more packets at the start of the subsequent FEC window following a lost FEC data packet may also be lost. Thus in addition to relying on `fwnd` and the count of total packets received within an FEC window, you must also rely on the sequence numbers in arriving packets to determine when you have received an FEC-window full of data bytes. To that end, in addition to keeping track of lost packet offset, upon every data packet arrival, first check whether you have received an FEC-window full (or more) of data bytes without receiving any FEC packet. In which case, you need to reposition your FEC window by computing the start of the current FEC window, reset your count of packets received, and determine the next expected packet. This should take about 7 lines of code.

That's it for Task 2. It again takes less than 30 lines of code.

Testing Your Code

The provided reference implementation of both server (`reffecdb`) and client (`reffecimg`) run on the CAEN eecs489 servers. You should test your code with small-ish mss, say 2072 bytes, and a large enough receiver window size, say 200, to see how FEC can reconstruct lost packets. You know your FEC is working if you see an empty strip of the displayed image "patched/filled in" out of order. You should also play with the server's drop probability and observe how higher drop probability causes more gaps/streaks in the displayed image that our simple FEC cannot reconstruct.

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. Both your client and server must interoperate with the reference server and client respectively.

Your "*Lab6 files*" comprises your `fec.cpp`, `netimg.cpp`, and `imgdb.cpp` files.

To turn in your Lab6:

1. Submit the SHA1's of your *Lab6 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 *Lab6 files* by pointing your web browser to Course folder and navigate to your `lab6` folder under your uniqname. Or you can `scp` the files to your `lab6` folder on IFS: `/afs/umich.edu/class/eecs487/w15/FOLDERS/<uniqname>/lab6/`.
   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.