EECS 489 PA3: Reliable UDP with FEC

This assignment is due on Friday, 3 April 2015, 10 pm.

Overview

In this programming assignment, you are to implement reliable UDP using Go-Back-N and cumulative ACK, with a simple XOR-based FEC used to improve performance when network loss rate is low. The specification of this assignment relies heavily on Labs 5 and 6. You may also find the PA3 walk-through lecture slides helpful. The assignment consists of the following graded tasks.

Graded tasks (100 points total)

1. Transmission using datagram and scatter/gather buffer management (15 pts)
2. Go-Back-N with cumulative ACK (30 pts)
3. FEC without Go-Back-N (25 pts)
4. FEC with Go-Back-N (30 pts)
5. Writeup

The support code for this assigment can be downloaded from the Course Folder. Whereas both Labs 5 and 6 are open-loop systems, the use of ACKs to clock data transmission in this assignment makes the system a closed-loop system. Given the fundamental difference between the two approaches, you've been provided with a new version of `imgdb.h`, `imgdb.cpp`, `netimg.h`, `netimg.cpp`, and `Makefile`. You'll need the other files, such as `socks.cpp` and `fec.cpp`, from Labs 5 and 6 not included in the support tar ball. Save your Labs 5 and 6 files before overwriting them with the version from this assignment. As usualy, the provided `Makefile` builds a `netimg` client and a `imgdb` server. To avoid overwriting the reference implementations from Labs 5 and 6, however, the provided reference implementations for this assignment are called `refgbnimg` for the client and `refgbndb` for the server.

Aside from the updated `netimg.h`, you are not required to use the provided support code. Should you choose to use the provided PA3 support code, you could copy over the relevant code from your Labs 5 and 6, or you may choose to follow the instructions in the new support code to modify your Labs 5/6 files. If you decide to write your own code from scratch, you're encouraged to review the comments and instructions in the provided support code, they form part of the specification for this assignment.

Assumptions

We will make several simplifying assumptions in line with Labs 5 and 6:

1. There is a single client and server pair in communication.
2. Image file transferred can not be larger than 2 GB.
3. There is no loss detection of the query packet sent from the client to server. If the query packet is lost, user would have to manually timeout, and terminate and restart the client.
4. The server will try at most 3 times to send the image dimension in reply to client's query. If the transmission is not ACKed after 3 tries, or if the ACK is malformed, the server will assume the client has terminated and will not transfer the image. It simply goes back to waiting for a query from client.

5. The client will not be terminated until the full image has been transferred and a `NETIMG_FIN` packet has been sent by the server. If the client is terminated before the server sends a `NETIMG_FIN` packet, the server could be left in an undeterministic state and subsequent client query may not be served correctly.
6. We associate a sequence number with each byte of data and the sequence number of a packet or segment is the sequence number of the first byte in the packet/segment.
7. We will always start a transmission with initial sequence number of 0.
8. We use cumulative ACK. ACK(*n*) means all bytes from sequence number 0 to *n-1* have been received and the receiver is waiting for sequence number *n*.
9. Instead of discarding out-of-order packets, we ACK and display them.
10. We don't estimate round-trip time. Instead we simply set the retransmission timeout to `NETIMG_SLEEP` secs and `NETIMG_USLEEP` microsecs. We assume that the manually set timeout is large enough for a window-full of packets to be acknowledged if there were no dropped packets. By default the timeout is set to 1.5 secs in `netimg.h`, this allows for a visual differentiation of error recovery by FEC or by ARQ. If you get tired of waiting for retransmission to kick in, set the timeout to a smaller value. On local host, a retransmission timeout of 500 ms is usually sufficient. Running the server and client on CAEN eecs489 hosts when connected over ADSL or cable, you may have to set retransmission timeout to 20 seconds or longer.

Your Tasks

1. Transmission using datagram and scatter/gather buffer management

This part of the assignment is covered in Lab 5. You may re-use both the support code and your code from Lab 5. This part of the assignment allows you to observe the role the receiver buffer plays in datagram transmission. It allows you to observe what happens when there's no flow control. You can change the size of the receiver buffer by modifying the receiver window and/or the maximum segment size. You can also modify the packet drop probability of the server and observe what happens when there is no error recovery at the transport layer. This part of the assignment also requires you to use gather write for transmission of large file and the corresponding scatter read on the receiver side. Not only does the use of scatter write help us visualize lost packets, it saves us from having to maintain a separate buffer for FEC window. If you decide to use the provided support code, you'll need to migrate your Lab 5 code over. Search for the string "Lab 5" to find the places in `socks.cpp`, `imgdb.cpp`, and `netimg.cpp` where your Lab 5 code must go.

2. Go-Back-N with cumulative ACK

This part of the assignment have you add flow control with sliding window, and retransmission with Go-Back-N, using cumulative ACK. If you decide to build on your own lab source code instead of the support code, you should build off your Lab 5, not your Lab 6, code for this task.

Task 2.1. Session Initialization

First we implement the session initialization handshake which consists of the client sending an `iqry_t` packet to the server. This has been implemented for you in `netimg_sendqry()`. On the server side, the server waits for a valid query message from a client. As a previous client may have left some ACK packets on the server's receive buffer, the server continues to grab these off its buffer, by repeatedly calling `imgdb::handleqry()`, until a valid query packet is received. You don't have to write any new code for this task.

To send a queried image back to the client, the server first sends an `imsg_t` packet by calling `imgdb::sendpkt()`. Wheres in Labs 5 and 6 `imgdb_sendpkt()` simply sends the packet to the client, here you should wait for an `NETIMG_ACK` packet after sending the `imsg_t`, up to a timeout time. If you time out without receiving an ACK, re-send the packet and wait for ACK again up to `NETIMG_MAXTRIES` times. When a properly formatted ACK has been received, `sendpkt()` returns 0 to caller. This takes about 15 lines of code, including Lab 5 `imgdb::sendpkt()` code. The caller will check that the ACK packet has sequence number `NETIMG_SYNSEQ`, so don't forget to convert the `ih_seqn` field in the ACK packet to host byte order.

Back on the client side, you must add code to `netimg_recvimsg()` to send an ACK back to the server upon receiving an `imsg_t` packet. As described above, the server would be expecting an ACK packet of type `NETIMG_ACK` and sequence number `NETIMG_SYNSEQ`. Here you may also want to initialize any state necessary to implement Go-Back-N on the client side. This task should take about 5 lines of code.

In total this task takes about 20 lines of code that should be familiar to you already. Search for the string "Task 2.1" to find where in `netimg.cpp` and `imgdb.cpp` you need to add code to complete this task.

Task 2.2. Go-Back-N server side

Now we're ready to send the image to the client. In `imgdb::sendimg()` first initialize any variables you need to keep track of your sliding window: size of the window, the first and last byte of the window, and any other variables you may need. Then we go into a loop sending the image data, waiting for ACK, and retransmitting the image data if necessary. To implement flow control, first update your estimate of the available space on the receiver's receive buffer based on the receiver's advertised window size, the amount of data you have sent, and the amount that has been acknowledged. We'll call this the "usable" window. Then while the usable window is larger than an mss and there's still data to send, send the data one segment at a time. Migrate your code from Lab 5 here to do the sending of each segment using `sendmsg()`. Also migrate the support code from Lab 5 that probabilistically drop a segment instead of sending it. You should update your sliding window variables for every segment you sent, including the usable window size.

Once you've sent out as many segments as the usable window allows, wait for ACKs with timeout, similar to what you did in `imgdb::sendpkt()`. If an ACK arrives before you time out, grab the ACK from the receive buffer, and update your sliding window variables as necessary. We will now opportunistically grab all the ACKs that have arrived instead of going back to wait for the next timeout. Everywhere else in `imgdb` we want the socket to be blocking. Only when we do this opportunistic grabbing of arriving ACKs, we don't want to block if no ACK has arrived. Instead of using `ioctl()` to toggle the socket blocking/non-blocking back and forth, we'll call the socket receive function with `flags=MSG_DONTWAIT`, which is equivalent to setting the socket non-blocking, but only for the call where the flag is set. Upon returning from the receive call, the socket will be put back to the mode it was on.

If you have received only one ACK, then your sliding window could have slide forward one segment. If multiple ACKs arrive, then your sliding window could have slide forward multiple segments. When you return to the top of the loop, your usable window size will then be updated for the next round of transmissions. Before you return to the top of the loop though, if you time out without receiving any ACK, you have experienced a retransmission timeout (RTO) and you need to invoke Go-Back-N and retransmit all segments starting from the one you're waiting acknowledgement for. All you need to do to perform retransmission is to reset your sliding window to start at the byte for which you're waiting for acknowledgement.

Finally, after you've sent the full image data and all the segments have been acknowledged, send out a

NETIMG_FIN packet with sequence number NETIMG_FINSEQ using the imgdb::sendpkt() function you wrote in Task 2.1. Recall that this function waits for an ACK and retries NETIMG_MAXTRIES times if an ACK doesn't return. In this case, if the ACK doesn't return, we simply give up after NETIMG_MAXTRIES tries and move on to serve the next client.

This task takes about 30 lines of code, not counting the code from Lab 5 interspersed among the new code. Search for the string "Task 2.2" in imgdb.cpp for places where your code should go.

Task 2.3. Go-Back-N client side

Meanwhile, on the client side, in netimg_recvimg() if an arriving packet is a data packet (type NETIMG_DATA), grab it off the socket buffer. If its sequence number is the one we're expecting, update our expected sequence number. In all cases, prepare an ACK packet with the correct type and the expected sequence number. If we receive a NETIMG_FIN packet, grab it off the socket buffer and prepare an ACK packet with sequence number NETIMG_FINSEQ. If we have an ACK to send, either send it now or drop it with certain probability (to simulate lossy link) as you do with data packet on the server side.

This task should take about 10 lines of code, not including code from Lab 5 interspersed herein. Search for string "Task 2.3" in netimg.cpp to find places you need to put your code. That is all for Task 2. You should now be able to build imgdb and netimg to test your implementation.

3. FEC without Go-Back-N

This part of the assignment is covered by Lab 6. You may re-use both the support code and your code from Lab 6. This part of the assignment have you add forward error correction (FEC) to datagram transmission. If you have not done Lab 6, you may want to do this task using Lab 6 support code instead of adding FEC directly to the code you have been working on in the previous task. Adding FEC to datagram transmission that also does flow control with the sliding window protocol and retransmission using Go-Back-N is more complicated and actually forms the next task. Once you have a working Lab 6, you can migrate your Lab 6 code over to your code above that has Go-Back-N implemented. Search for the string "Lab6" in imgdb.cpp, netimg.cpp, and fec.cpp in the Lab6 support code to find where your code must go, see also the description for the next task for more detailed explanation. Unfortunately you can't test your migrated FEC code until you have completed the next task. To unit test this task, you should implement and test Lab6 separately.

4. FEC with Go-Back-N

The FEC we implement in this assignment can only patch up one lost segment per FEC window. When the network loss rate is low, the FEC can help improve performance by preventing the sender from stalling due to retransmission timeout and can reduce network utilization by obviating retransmission of packets that are out of order simply due to the lost of a single packet. When network loss rate is high or when we have multiple losses per FEC window, however, we must still rely on Go-Back-N for the correct operation of our reliable UDP.

Task 4.1. FEC with Go-Back-N server side

In imgdb::sendimg(), when you initialize your sliding window variables, initialize your FEC window variable(s) as necessary also. Then within the loop where you send each segment off to the client, before you send off each segment, update your FEC variable(s) and FEC data packet by migrating your Lab 6 code here. Then after you've sent off each segment, if you've accumulated an FEC window full of data or if you've just sent the last segment of the file, send out your FEC data packet also. So far, all you've been asked to do here is to migrate your Lab 6 code over, inserting them at the appropriate places. The only new code you need to write for this task is to reset your FEC variable(s) and FEC window when you time out waiting for an ACK. When the server experiences a retransmission timeout, it goes into Go-Back-N mode and retransmits all packets, starting from the sequence number it is waiting for acknowledgement. When entering Go-Back-N, the server should also resets its FEC window to start at the retransmitted sequence number. This should be no more than one or four lines of code. Search for "Task 4.1" in `imgdb.cpp`.

Task 4.2. FEC with Go-Back-N client side

The single most complicated aspect of using FEC together with Go-Back-N is the interaction between packets that are already "in flight" following a lost packet and the FEC window. Recall that in Lab 6 we rely on a simple count of how many packets have been received within an FEC window to decide whether we can use FEC to patch up a single lost segment. You can imagine how retransmitted packets could mess up our count. Since FEC can only patch up a single lost packet within an FEC window and therefore, hopefully, avoid a retransmission timeout, when we **do** experience a retransmission timeout, more than likely we have lost more than 1 packets within an FEC window, so we should simply "ride out" the packets already in flight and "deactivate" FEC until the retransmitted lost packet has been received. To that end, you should keep a global variable that the function `netimg_recvimg()` can use to determine whether it is in Go-Back-N mode. When in Go-Back-N mode, the function doesn't update any FEC variables. It gets out of Go-Back-N mode when it receives the retransmitted lost packet. In the following we discuss how the client can put itself into Go-Back-N mode.

When `netimg_recvimg()` receives an `NETIMG_FEC` packet, if the client is not in Go-Back-N mode, check if the client has **lost one single packet** within the FEC window. If so, use your Lab 6 code to reconstruct the packet and send back an ACK tagged with the sequence number carried in the FEC data packet. If **no packet has been lost**, simply "throw away" the FEC data packet and don't send any ACK back. If **more than one** packets have been lost, put the client in Go-Back-N mode. As an optimization, if you **lose more than one consecutive** data packets, you are **REQUIRED** to immediately put the client in Go-Back-N mode. In all cases, reset the FEC window packet count and reset the FEC window to start at the next expected sequence number (which may be the sequence number of the lost packet). When the client is in Go-Back-N mode, if a data packet with the next expected sequence number arrived, take the client out of the Go-Back-N mode. This task requires about 10 lines of code.

As in Lab 6, a `NETIMG_FEC` packet could be lost. The only way to detect this is when handling a `NETIMG_DATA` data packet. If we have received an FEC-window full of consecutive data and the next packet received is another `NETIMG_DATA` packet instead of a `NETIMG_FEC` packet, we know that we have lost an FEC packet. In this case we haven't lost any data packet in the previous FEC window, so we don't mind losing the FEC packet and we can simply slide the FEC window forward to the next window.

Otherwise, if we have received less than an FEC-window full of data but the gap between the start of an FEC window and the received segment is larger than an FEC-window full of data, we have lost both the FEC packet and one or more segments within that FEC window. There's no way for us to recover the lost segment(s) and Go-Back-N will be triggered. We should now "ride out" the "wave of retransmitted packets" by putting the client into Go-Back-N mode and not rely on FEC until the expected segment has been retransmitted and received. At which point, we reactivate FEC, restarting the FEC window at the retransmitted segment. This takes about 8 lines of code.

When not in Go-Back-N mode, we keep track of packet received within the current FEC window as in

Lab 6. You should be able to implement handling of the lost of FEC data packet with 8 new lines of code, in addition to your Lab 6 code for the purpose.

Finally, you must detect when you're at the last FEC window of a transmission. The last FEC window of a transmission may be smaller than the FEC window of the rest of the transmission. So everytime you slide your FEC window forward, if you're at the last FEC window, reset your FEC variables such as `fwnd` and other relevant variables to fit the last FEC window. This should take about 3 lines of code.

That's it for this task. It requires about 30 new lines of code. Search for "Task 4.2" in `netimg.cpp`. Task 4 in total takes about 35 lines of code.

Can we do better?

We could use Reed-Solomon code instead of the simple XOR. That will allow us to reconstruct multiple loss packets within an FEC window. We could also use Selective-Repeat instead of Go-Back-N and retransmit only lost segments. However, the retransmission code for Selective Repeat will be a lot more complicated. Instead of simply keeping track of the first lost packet, we will need at least a bitmap scoreboard as large as the receiver window to keep track of segments received and we would have to modify the protocol to communicate this scoreboard to the sender. As for Reed-Solomon code, it's both more complicated and easy. Easy because there are several open-source Reed-Solomon libraries you could use. You're welcome to try to implement either or both of this if you're interested. (No extra credit though.)

Testing Your Code

Try to run your server with different drop probabilities. When the loss rate of a path is low (drop probability <0.05, roughly), the occasional lost packet can be patched up by FEC. When the loss rate is high (drop probability >.2, for example), we pretty much have to rely on ARQ, in this case Go-Back-N, to recover the error. Try playing with different `rwnd` and `mss` also. Remember that the size of the FEC window is a function of `rwnd`. To help you test, you may want to instrument your code to drop only data packets, not FEC nor ACK packets, or to drop only FEC packets, or only ACK packets, or not to drop more than one packet per FEC window, or to drop multiple packets per FEC window, etc.

Submission Guidelines

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. For example, the assignment asks you to use scatter/gather buffer management with your file transmission. If you turn in a working program that does file transmission without using scatter/gather buffer management and you do not inform the teaching staff about it, it will be 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 5 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 6. Unlike previous programming assignments, you are provided, free of charge, with a skeletal `imgdb.cpp` and `netimg.cpp` in this assignment.

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 should interoperate with the provided `refgbnimg` and `refgbndb` binaries in the Course Folder.

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 PA3:

1. Submit the SHA1's of your *PA3 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 *PA3 files* by pointing your web browser to Course folder and navigate to your `pa3` folder under your uniqname. Or you can `scp` the files to your `pa3` folder on IFS:
   `/afs/umich.edu/class/eecs487/w15/FOLDERS/<uniqname>/pa3/`.
   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.