Homework 4 Documentation

Author

William Montgomery

Copyright

None

University of Illinois at Chicago

CS 450 - Introduction to Networking

Spring 2014

About the project

The file transfer project is composed of two programs, client and server. The general idea is to allow the client to transer files and headers over UDP, and the server to process those messages. In this iteration of the project, I have implemented Selective Repeat.

Documentation

The project is documented with Doxygen tags. These tags allow HTML and RTF documentation to be generated. The HTML documentation can be found in the docs/html/index.html document. It is recommended that a web browser be used to view the documentation if possible.

Building the project

A makefile is included. The user can run 'make clean' and then 'make' to build both the client and server applications (client and server respectively). server saves files in the current working directory.

Running the server

The server can be run after building with './server'. It takes the following command line arguments:

- port - This argument specifies the port to connect to. The default is 54323.
- windowSize - This argument specifies the server's default window size.

The server will save files to the local filesystem if the correct flag is set in the header. The files are stored in the ./files/ subdirectory.

There is no built in method to gracefully exit from the server. The user is directed to use [control] + C to exit the program.

The server only handles one client at a time. If a client sends a message while another client is transmitting, the first client's message will be dropped. Ideally, I would create a hashtable to handle concurrent messages, with the keys computed from the sender's address, port, UIN, and transaction number. I chose not to because messages that are not fully sent would need to be cleaned up sporadically and dead messages would sit in the table until program exit. I feel this is out of the scope of this assignment.

Running the client

The client can be run after building with './client'. It takes the following command line arguments:

- server - This is the IP address of the final destination that the packet is headed to. The default is 127.0.0.1.
- server port - This is the port of the final destination that the packet is headed to. The default is 54323.
- relay - This is the IP address of an intermediate relay. The default is 127.0.0.1.
- relay port - This is the port of an intermediaate relay. The defaul is 54322.
- chance - This sets the following fields in the header: dropChance, dupeChance, garbleChance, and delayChance. This defaults to 0.

Please note that the parsing of these inputs is brittle at best. Any incorrect value will cause the program to exit.

The client is generally interactive, looping through and asking the user to enter information interactively using basic menus. Menu items can be selected by entering the number next to the desired option. The client records basic statistics regarding time and number of bytes sent, on a per message and on a per session basis.

Analysis

For the analysis of this project, I decided to break it down into three phases. In the initial phase, I compared the results from the Stop-and-Wait and the Go-Back-N implementations to the implementation with Selective Repeat with an arbitrary window size. In the second phase, I chose to analyze setting effects of differing chances for "bad" things to happen, i.e. dropped packets, duplicate packets, garbled packets, and delayed packets. For the final series of tests, I compared differing window sizes with a moderate amount of invalid packets to gauge the effect of window size on a route that has packet loss.

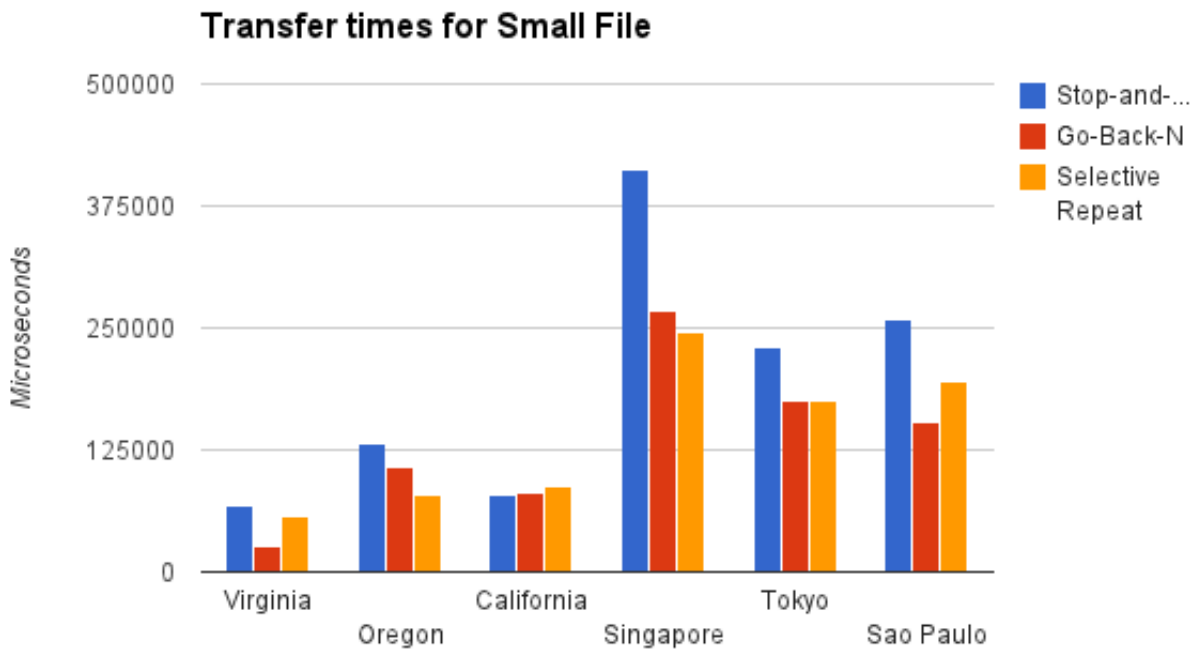
Stop-and-Wait vs Go-Back-N vs Selective Repeat

In this test, I sent a small file, a medium sized file, and a large file to the each of the following servers: Virginia, Oregon, California, Singapore, Tokyo, and Sao Paulo. I then sent a medium sized file through the relay in Virginia to the same list of servers. For reference, the small file was 30

bytes (1 packet), the medium file was 628361 bytes (176 packets), and the large file was 9437184 bytes (2634 packets).

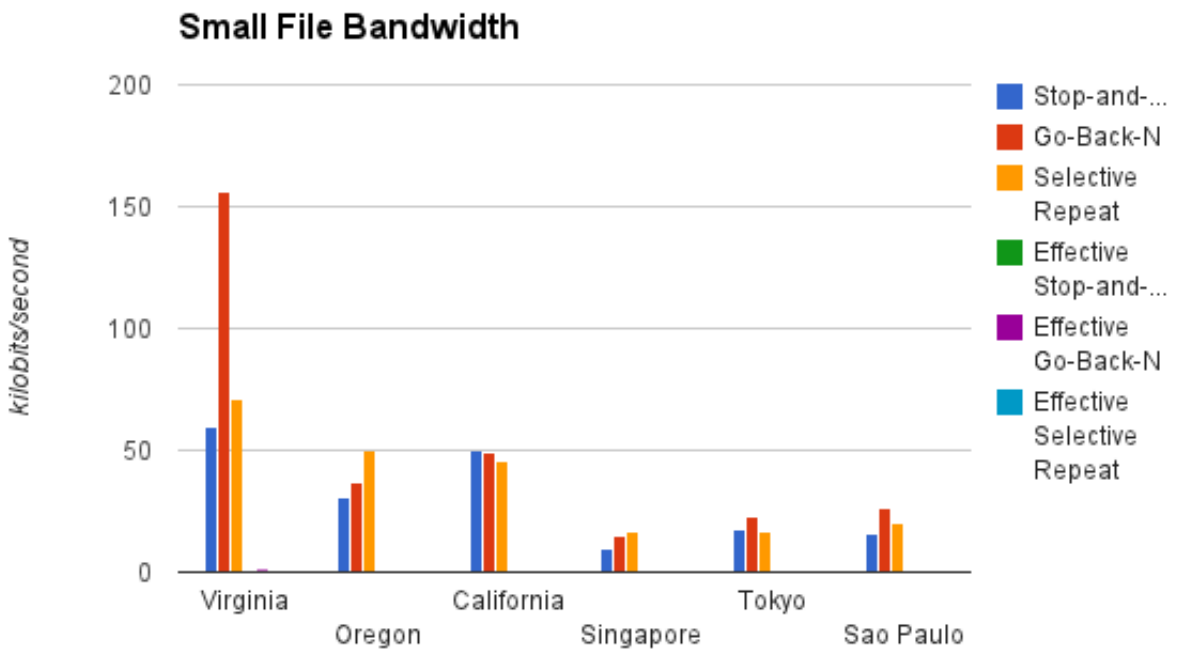
Small File Results

For the small file, the results were expected to closely match those of Stop-and-Wait and Go-Back-N as only a single packet was sent. The results in terms of transfer times are below (lower numbers are better):



As expected, the results were similar to Stop-And-Wait and Go-Back-N. Stop-And-Wait seemed to be slowest in most instances, but the differences could be attributed to network congestion.

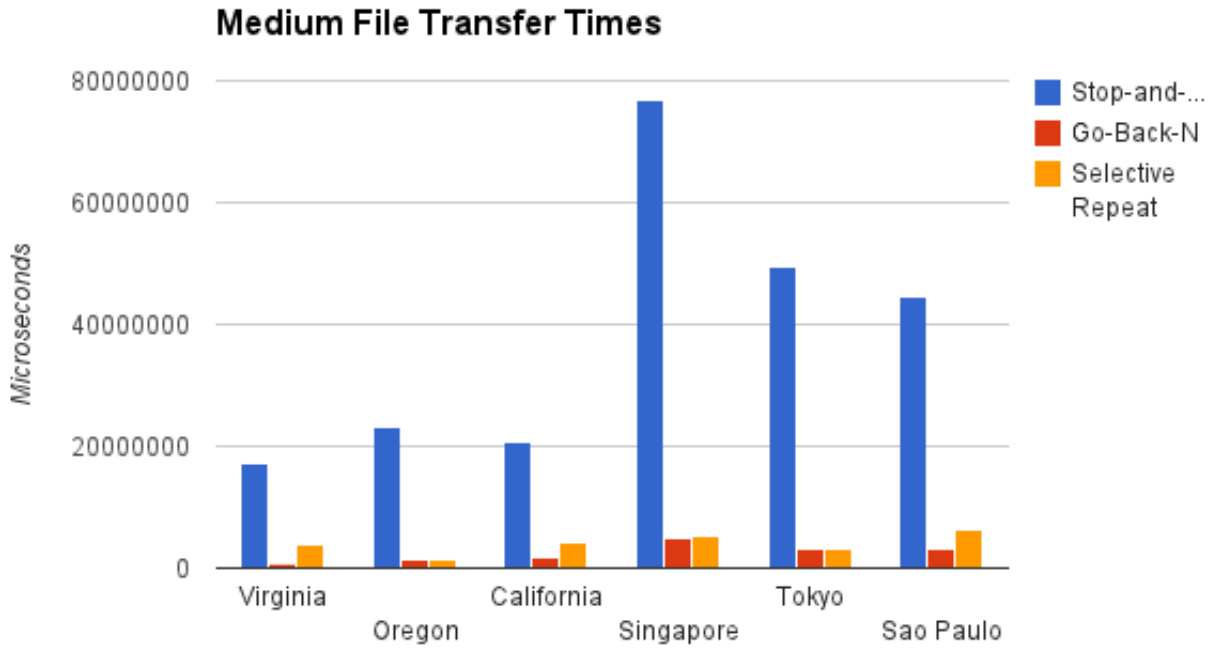
In terms of bandwidth, I took two measures. The first measure was the actual bandwidth used, not including resent packages. This is kind of a theoretical maximum bandwidth that the client could utilize if there was no overhead from the CS450 header and all of the packets were completely filled with data. This was calculated as (number of packets * 4096/transfer time in microsecond)*1000000/2^10 The second measure was the effective bandwidth. This is calculated as (file size / transfer time in microseconds) * (1000000/2^10). Both are calculated in kilobits/second. The results for the different servers are listed in the chart below.



As can be seen, the bandwidth utilization is about the same for all the servers, except for Virginia. This is as expected, and I would consider the results from Virginia to be suspect. The more interesting thing to note is that the effective bandwidth barely registers on this graph. This makes sense, as we are introducing a lot of overhead to send 30 bytes of data. The packet size that we are sending is a total of 4096 bytes. It should be noted that if we wanted to increase this effetive bandwidth utilization, we would want to decrease the size of the header and allow for packets of different size.

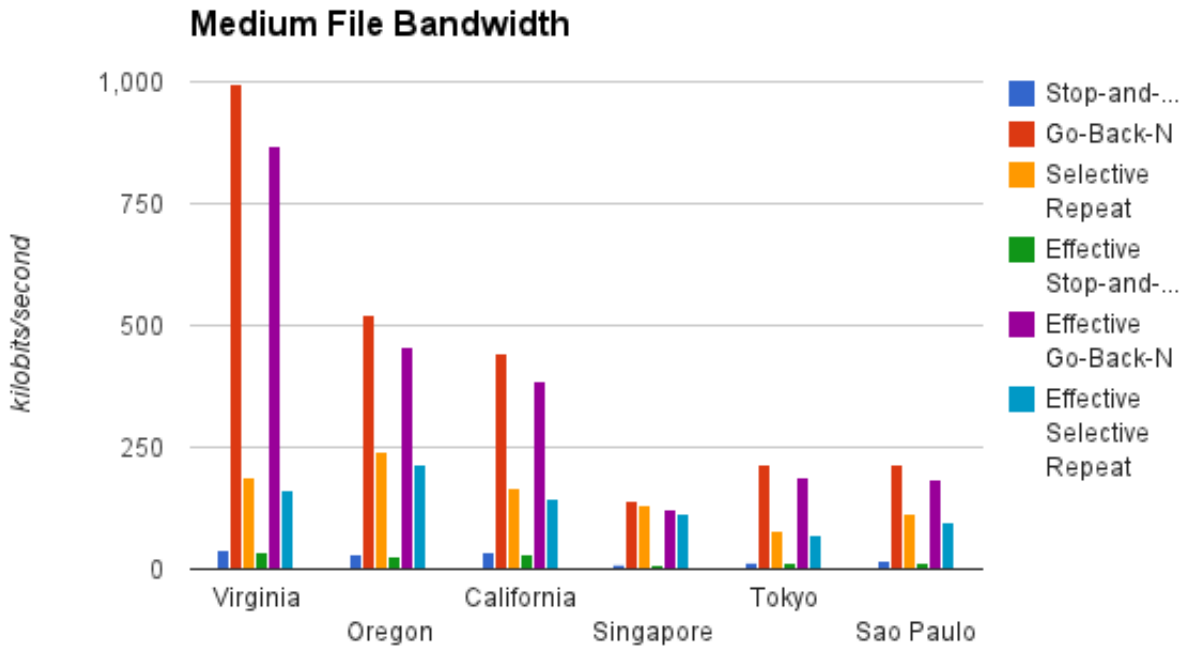
Medium File Results

For the medium file, it was expected that the transfer time for Selective Repeat would be better than Stop-And-Wait. It was unclear whether the transfer time would be better than Go-Back-N or not. Go-Back-N has an advantage that if an ack gets lost, it is likely that an ack will follow that is not lost that will supercede the lost ack. This is not the case with Selective Repeat, which requires an ack for every packet. If an ack gets lost, the client must timeout on that packet. This has the potential to stall the sending of packets.



The results show that Selective Repeat does much better than Stop-And-Wait, and does almost as well as Go-Back-N. It is interesting to note that Selective Repeat does take at least as long as Go-Back-N in every test.

The results in terms of bandwidth measures are included in the graph below (larger values are better):

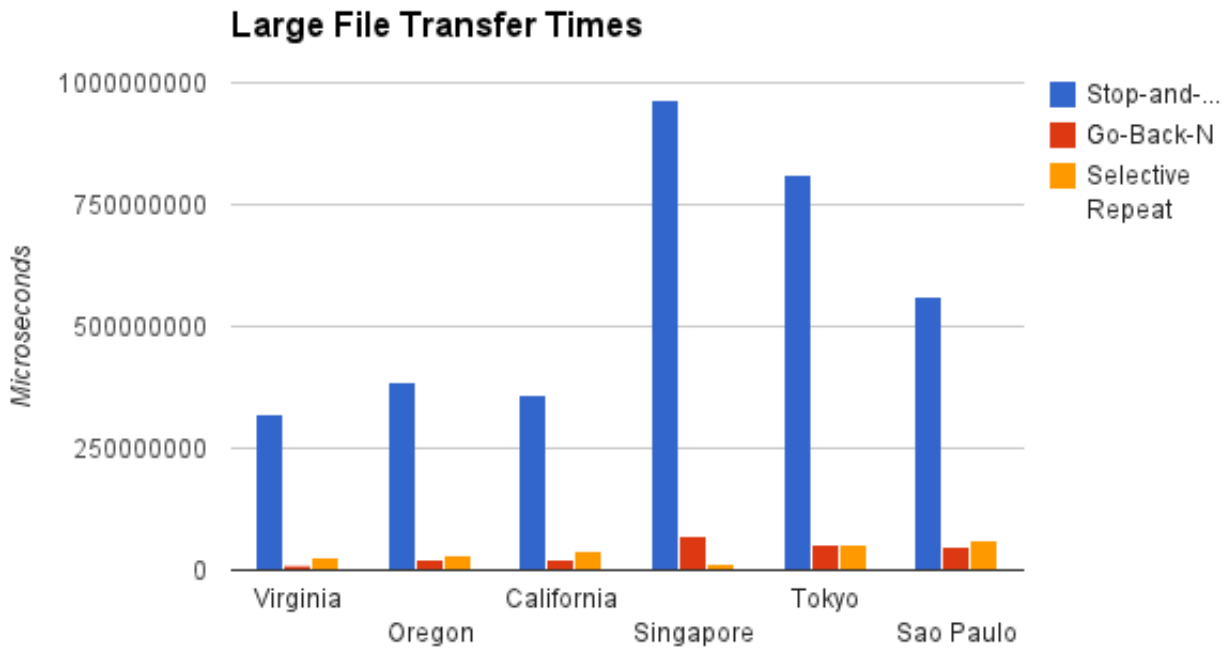


As can be seen, Go-Back-N was the clear winner in terms of bandwidth utilization. The timeouts

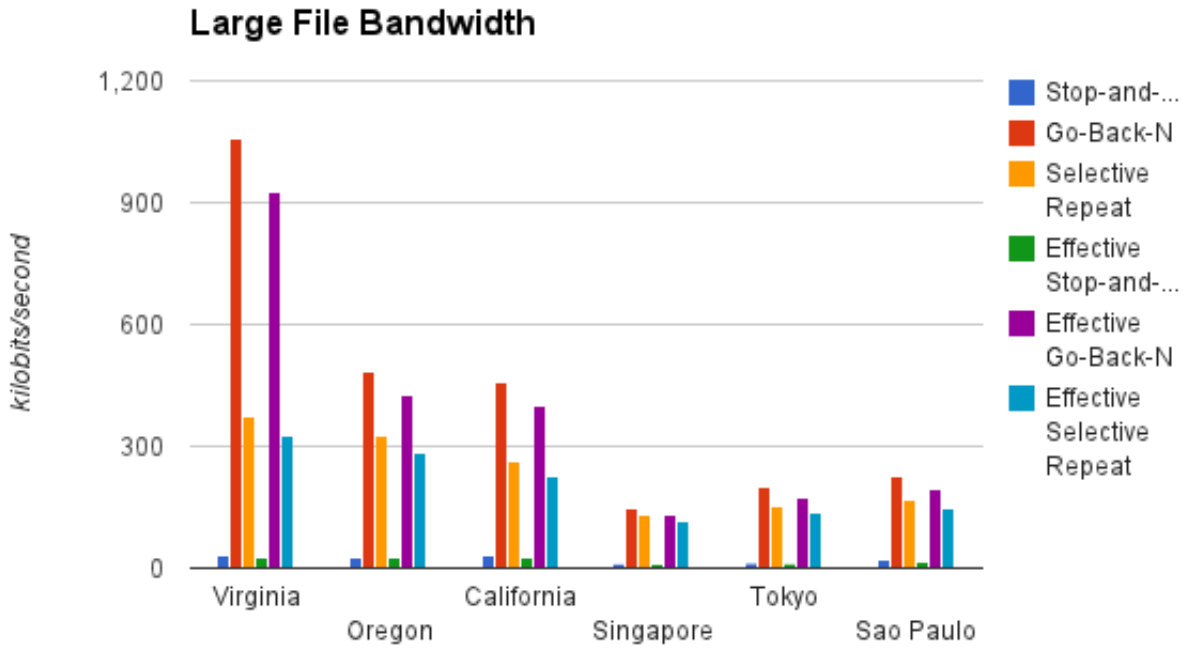
on the sent packets was clear. It should be important to note that I changed the algorithm for receiving acks, which might explain the increase in performance with Go-Back-N. Go-Back-N sends out the whole window before checking for an ack. This limits the potential window size, as the buffer in Linux fills up with 4k packets after about 32 packets. For Selective-Repeat, I check after every packet to see if there is a waiting ack. If so, it breaks out of sending to receive the ack. This adds considerable overhead, but allows a larger window size to be used. In a real implementation, the ack size would be much smaller than 4k. In this case, I would recommend sending out the window size and then reading in packets as long as the transmission time for a window is roughly equal to the round trip time.

Large File Results

The results for the large file in terms of transfer time are listed in the chart below (smaller values are better):



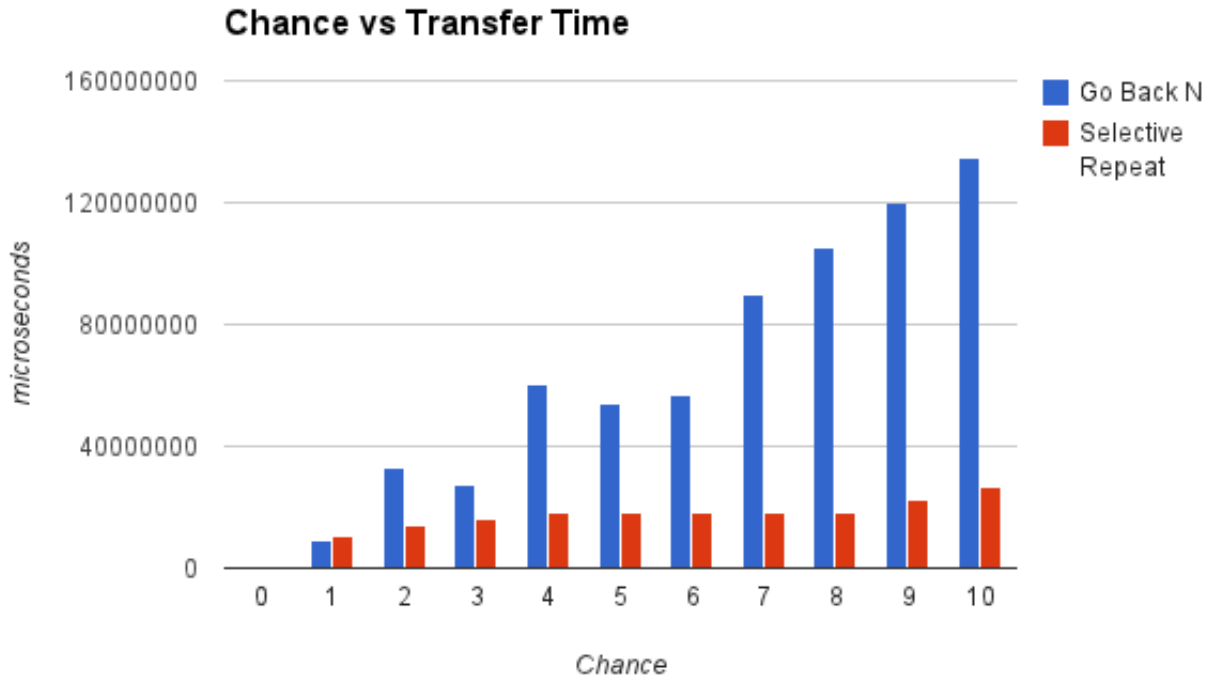
These are similar to the results from the medium sized file, except that Selective-Repeat beats Go-Back-N in the transfer to Singapore. These results are suspect, as the test to Singapore was performed on a different network than the rest of the tests.



These results are similar to the results from the medium file, except that the bandwidth utilization is closer to Go-Back-N for servers that are a long distance away. I imagine that if I were to utilize the bigger windowsize that my Selective Repeat implmentation affords, Selective Repeat would outperform Go-Back-N.

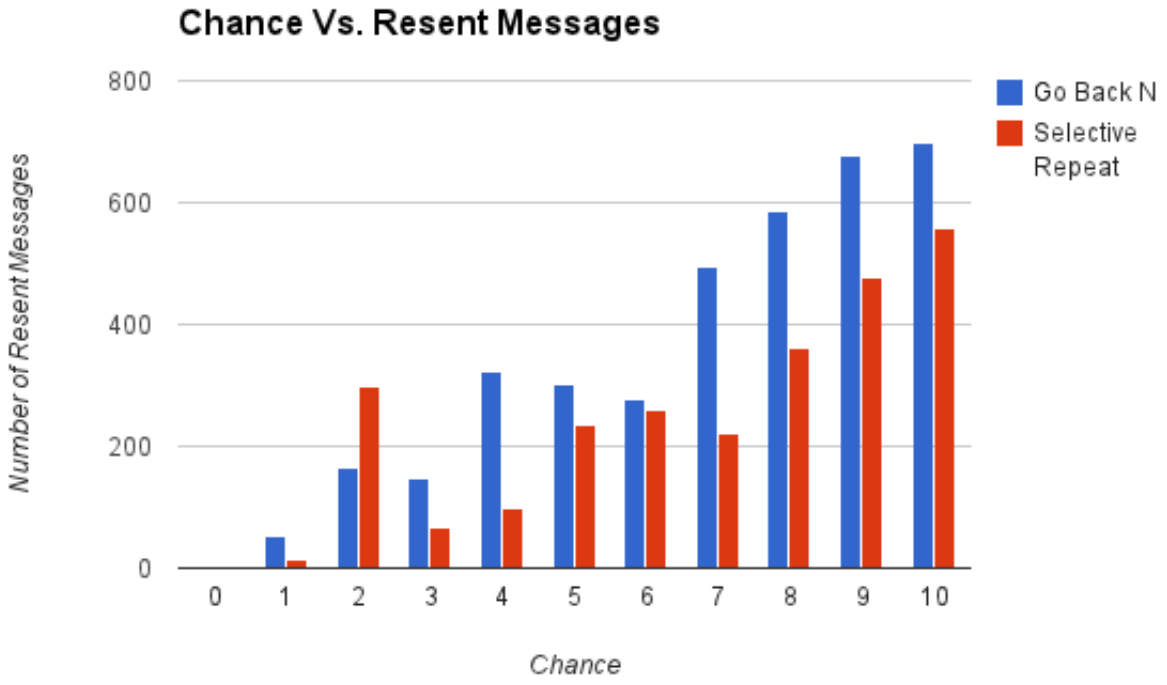
Chances

For this analysis, I sent a medium file through a relay with differing values for all of the "chance" fields in the header (dropChance, dupeChance, garbleChance, and delayChance). In each trial, I chose a value from 0 to 10 inclusive and set each of the chance fields to this value. A value of 0 indicates that there is no chance of the relay doing anything bad to the packets. A 10% chance is significantly more than a 10% chance of something bad happening, because there are 4 independent actions that can happen. It should be noted, that resent packets are also liable to be affected by the relay. The results in terms of transfer times are shown in the chart below (lower numbers are better):



It can be seen that increasing the chance also increases the transfer time. There is a slight anomoly around values of N of 3 and 9, but this is likely just the randomness of the chances creeping in. I would expect with larger files or more trials that there would be a steadier increase in transfer times. It is interesting to note that Go-Back-N increases at a much greater rate than Selective Repeat, which remains relatively constant. I believe this is due to the fact that the implementation of Go-Back-N throws away good packets if they are out of order. This increases the chance that the resent packets will be corrupted in transit.

The graph of the chance versus the number of resent packets is below (smaller is better):

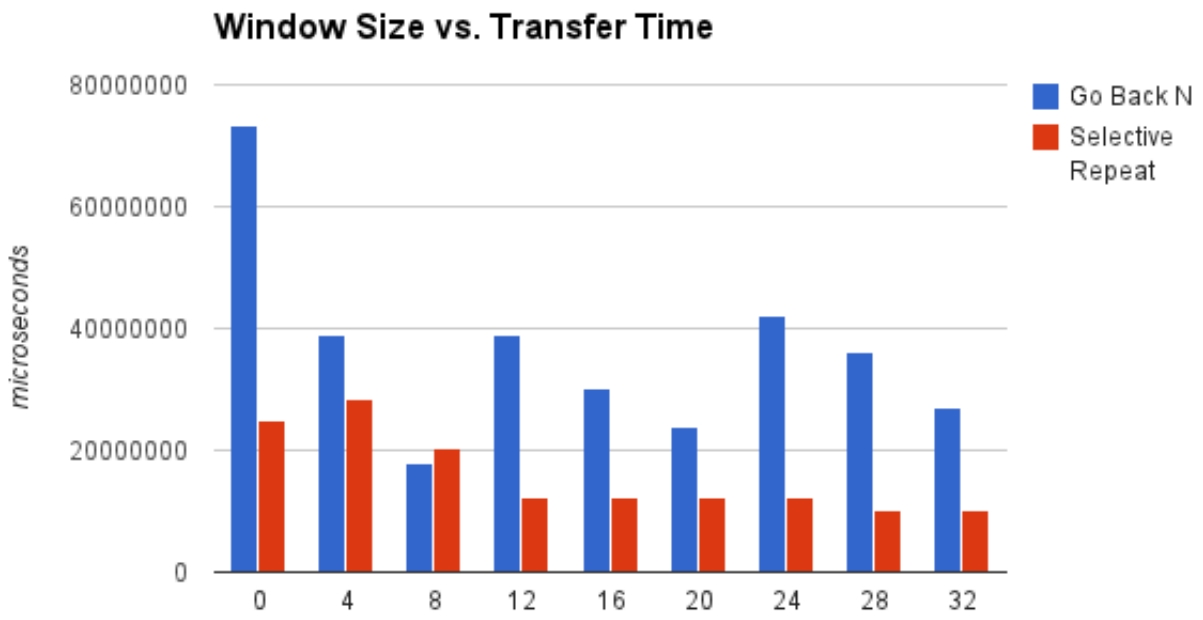


It can be seen that Selective Repeat generally beats Go-Back-N in terms of number of packets resent on a lossy connection.

Window Size

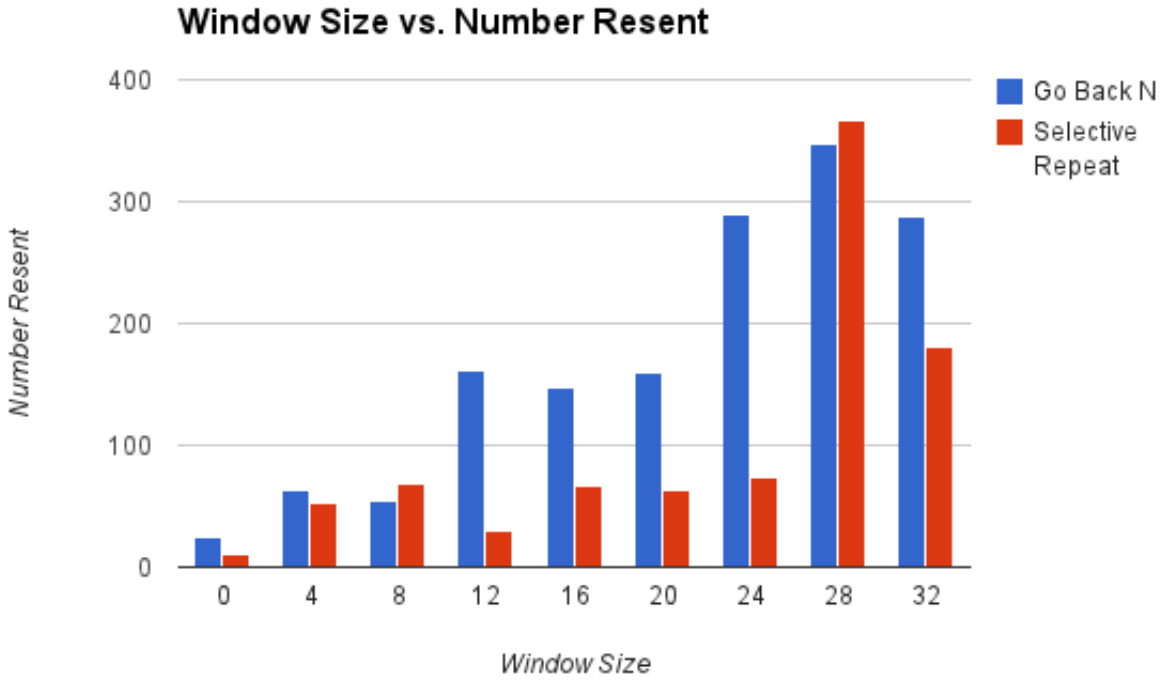
In these series of tests, I compared sending the packets out with different window sizes. I started with a window size of 0 (Stop-and-Wait) and increased the window size by 4 to a maximum of 32. I used a value of 2 for all of the chances. I should note that the Go-Back-N implementation had difficulty in sending a packet size of greater than 32. I believe that this is due to the fact that in my implmentation, I sent all unsent packets before trying to read an acknowledgement. I believe that when the window size got too large, I would receive acks until I filled up the buffer at which point the acks would get dropped. To correct this, I believe that I could check to see if there is data to be read in between sending packets. If so, I could read and process the ack, so that the buffer doesn't get filled up so quickly.

The results in regards to transfer time are below (smaller is better):



It is clear that Selective Repeat is the winner in this test, as I simulated a lossy connection. Unlike the test for Go-Back-N, which was highly variable, the test for Selective Repeat showed a larger window size was better. There were diminishing returns though, and the tests with window size 12 to 32 were very similar. I expect that a longer round trip time would show that large window sizes are better.

The graph fro window size versus the number of resent packets is below (smaller is better):



With the exception of the results from the window size of 28, we can see that Selective Repeat generally sends much less repeat packets.

Summary

From these results it is clear that Selective Repeat is preferable in situations where packet loss is expected, although Go-Back-N perfoms better in situtations without much packet loss. The hybrid approach that TCP makes balances these two conflicting situations, to provide better real world performance than either of the two individually.
