Homework 3 Documentation

Author

William Montgomery

University of Illinois at Chicago

CS 450 - Introduction to Networking

Spring 2014

About the project

The file transfer project is composed of two programs, myClient and myServer. 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 Go-Back-N.

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. There is detailed

Building the project

A makefile is included. The user can run 'make clean' and then 'make' to build both the client and server applications (myClient and myServer respectively). myServer requires a directory called 'files' to be created, to store the received files. The makefile should create this directory if it does not exist.

Running the server

The server can be run after building with './myServer'. 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 './myClient'. 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

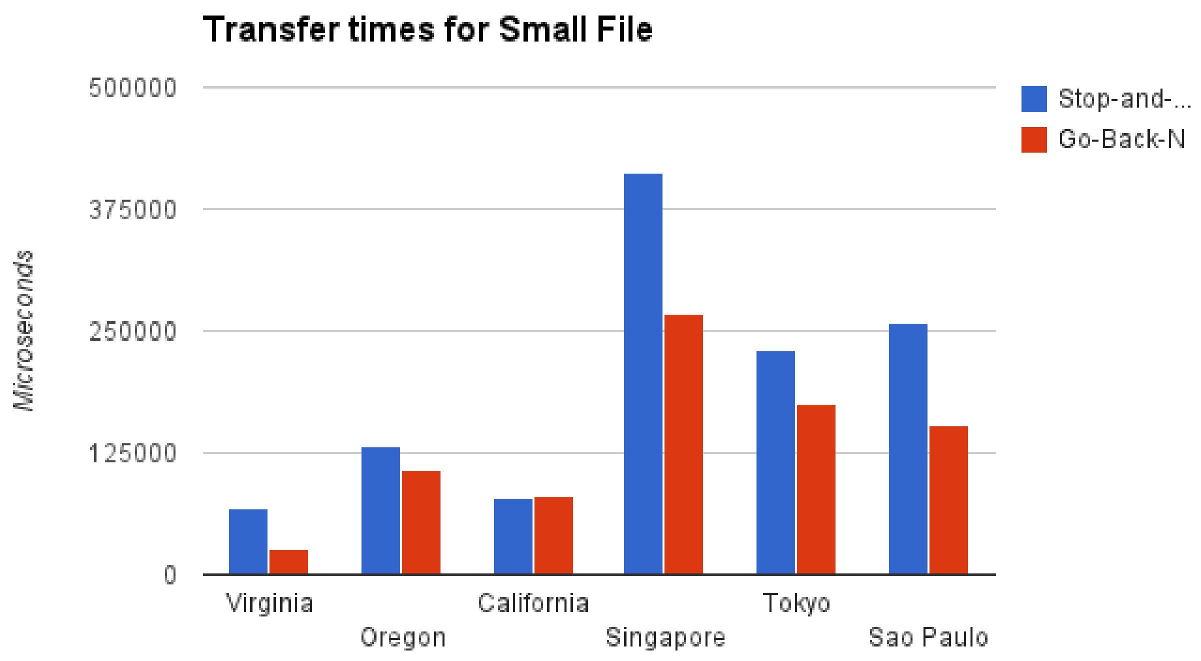
For the analysis of this project, I decided to break it down into three phases. In the initial phase, I compared the results from the Stop-and-Wait implementation to the implementation with Go-Back-N with an arbitrary window size. In the second chance, I chose to analyze setting effects of differing 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

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

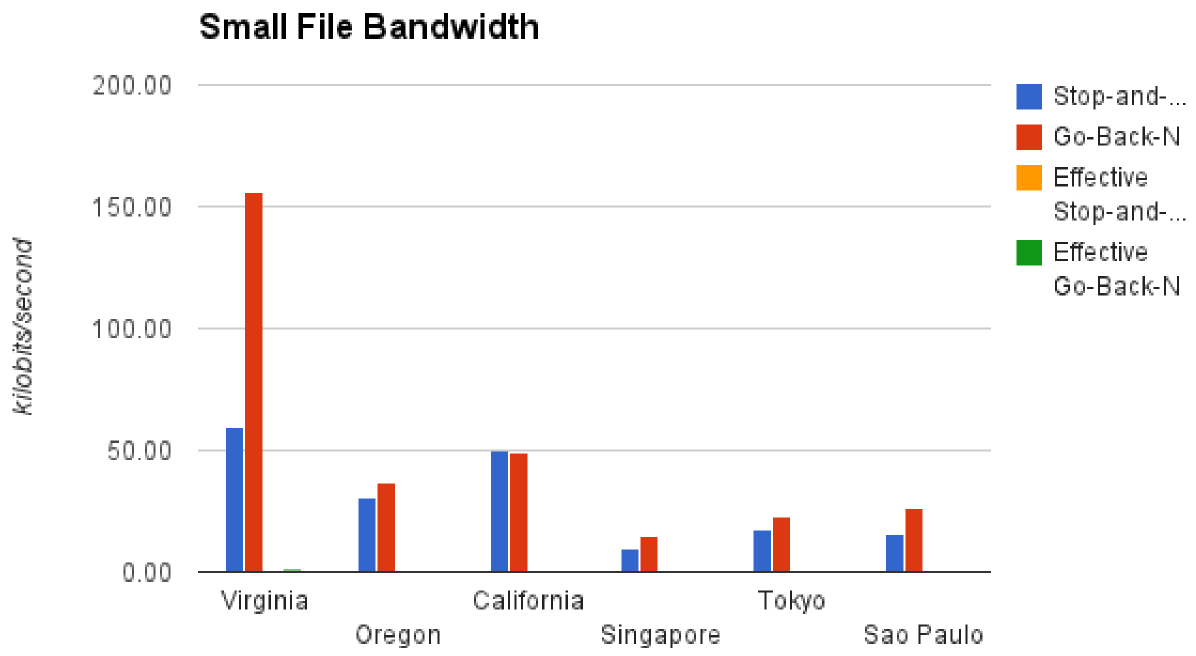
For the small file, the results were expected to closely match those of Stop-and-Wait as only a single packet was sent. The results in terms of transfer times are below (lower numbers are better):



Interestingly, for all of the servers, except for California, there was a significant decrease in the time needed send the packet. This can either be attributed to two factors. First, the network traffic could have been lower during the testing of Go-Back-N. Second, I changed a major portion of the implementation between Stop-and-Wait and Go-Back-N. In the Stop-and-Wait implementation, I set the socket to be non-blocking and just tried reading from it many times. This approach resulted in significant overhead. In the second approach, I used a blocking call to read from the buffer, but then implemented a timeout with the system call alarm. This was much more efficient.

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

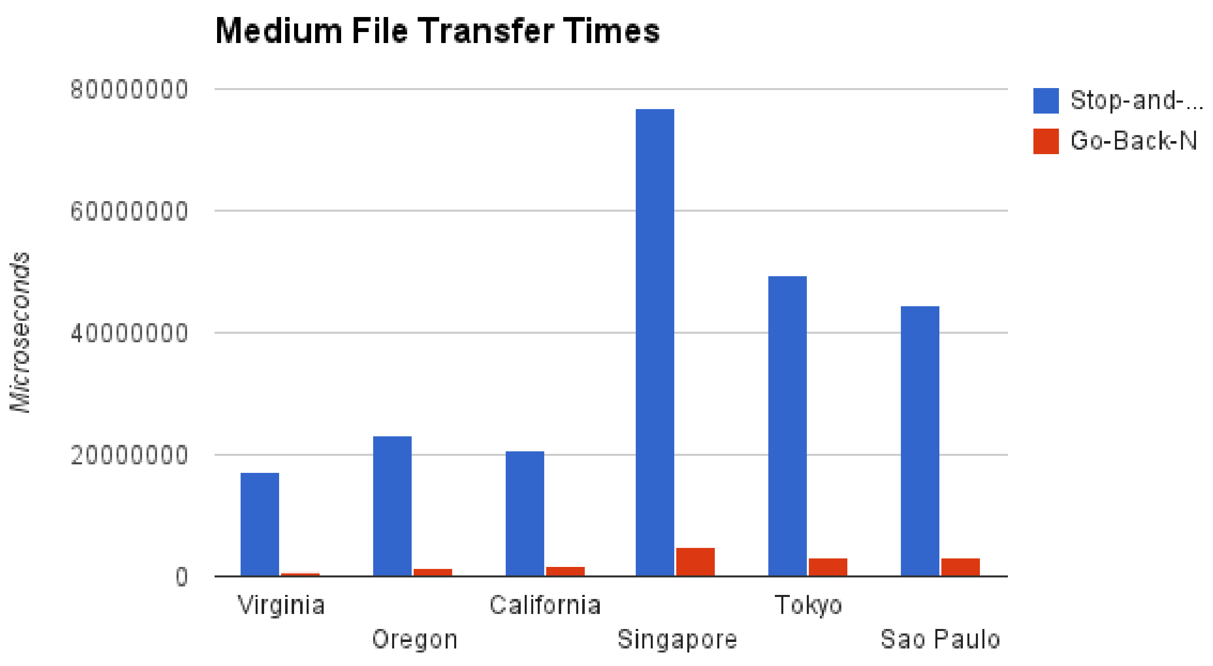
microsecond)*1000000/2^10 The second measure was the effective bandwidth. This is calculated as (file size / transfer time in microseconds) * (1000000/2^10). Both are calculated in kilobits/second. The results for the different servers are listed in the chart below.



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.

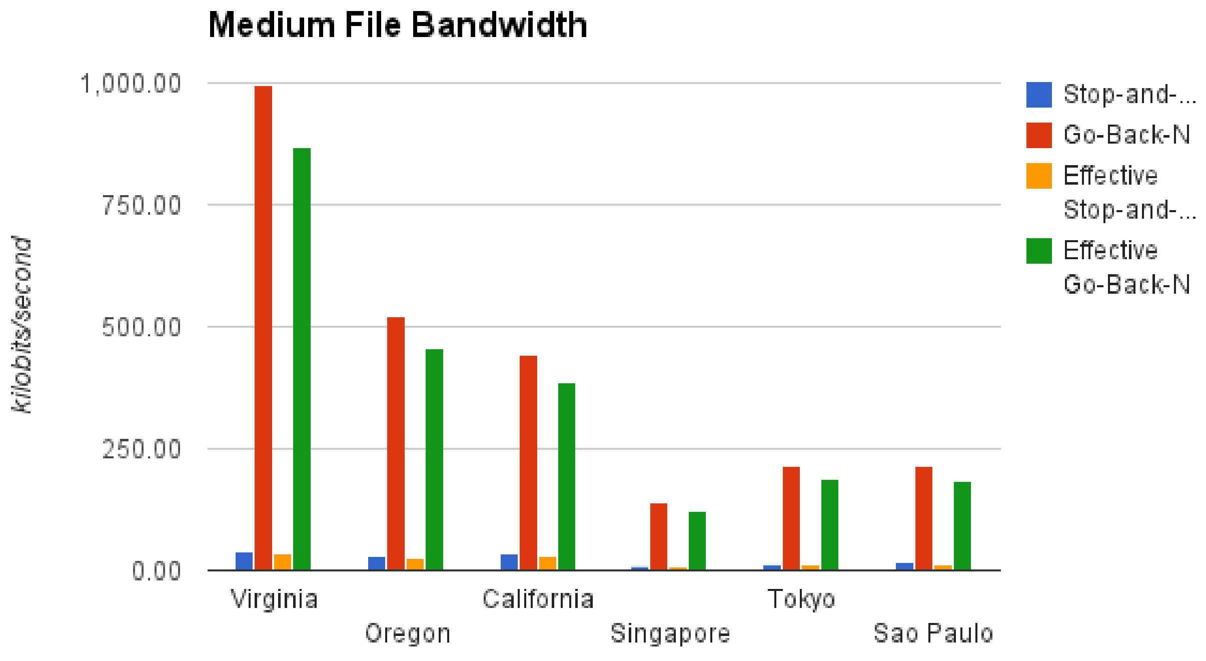
Medium File Results

For the medium file, Go-Back-N was expected to produce significant improvements over Stop-and-Wait, with the largest improvements coming from server with a large RTT (for example, Sao Paulo, Tokyo, and Singapore). The results in terms of transfer times are below (lower numbers are better):



Here we can see the improvement that Go-Back-N affords. On one end, we see the time from Virginia, which dropped from around 17 seconds to less than a second. At the other end of the spectrum, we see that the time from Singapore dropped from around 77 seconds to around 5 seconds. This is an order of magnitude improvement.

The results in terms of bandwidth measures are included in the graph below:

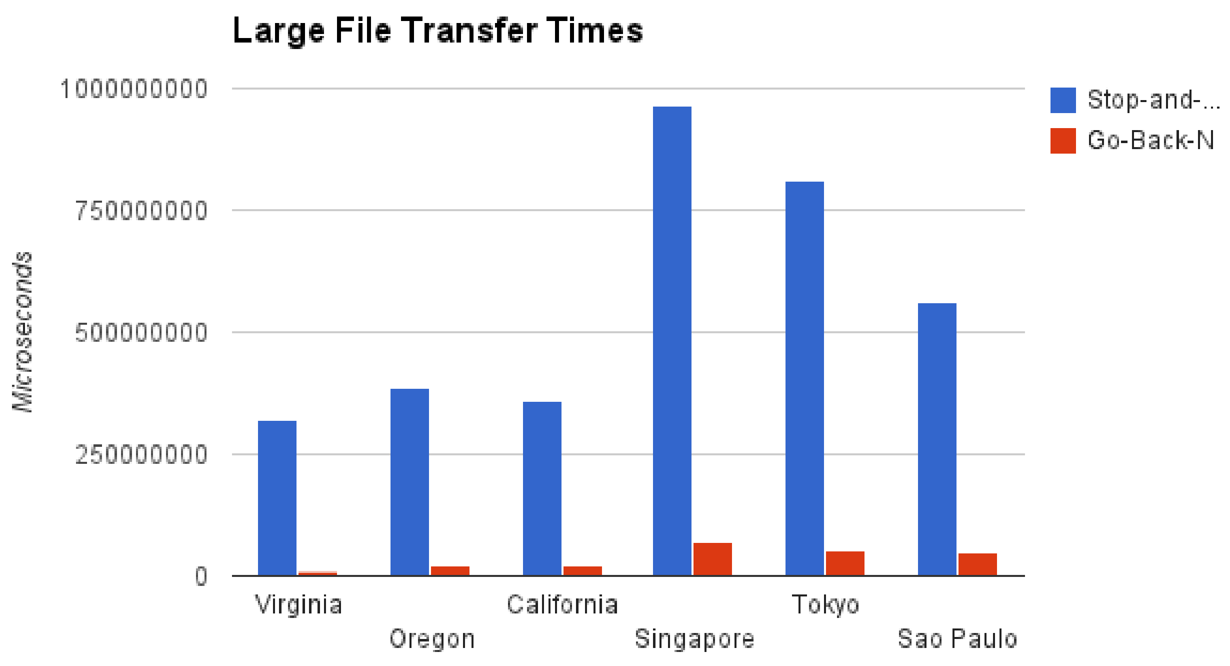


As can be seen, Go-Back-N had a significant increase in utilized and effective bandwidth for all

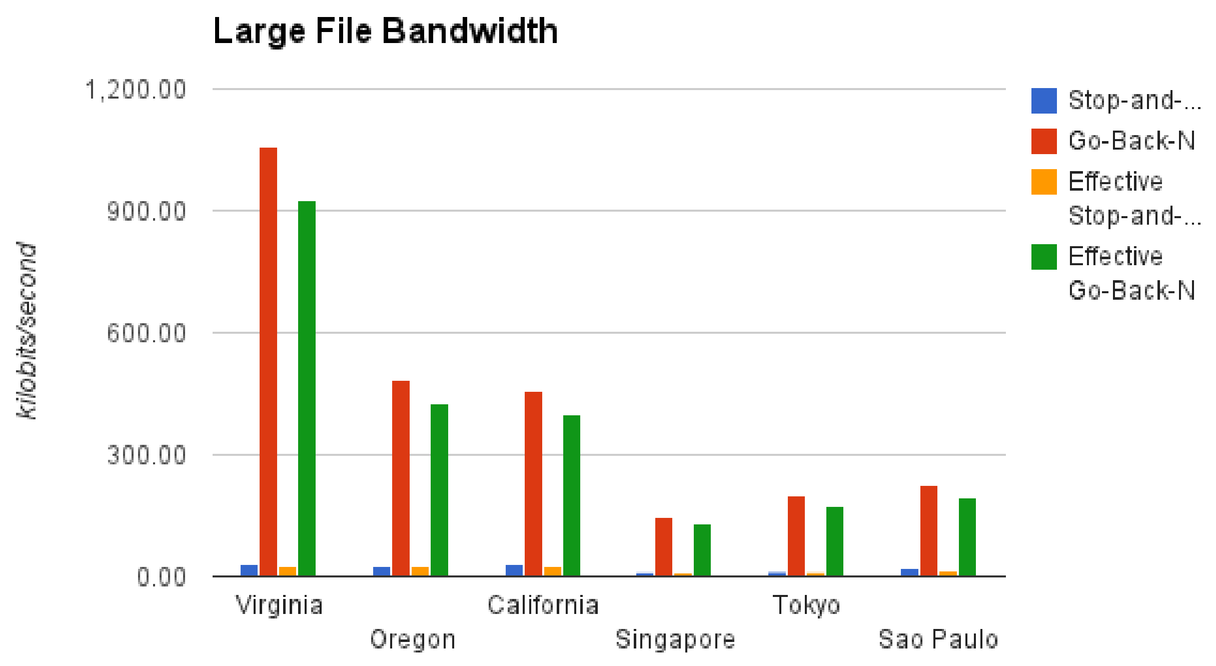
servers. It should be noted that the bandwidth for servers that were close was significantly greater than those far away. The window size was set at 16 for these tests. A larger window size would likely increase the performance of remote servers. In a real implementation, I believe that the timeout and window size would be adjusted based on the RTT between the servers.

Large File Results

The results for the large file in terms of transfer time are listed in the chart below:



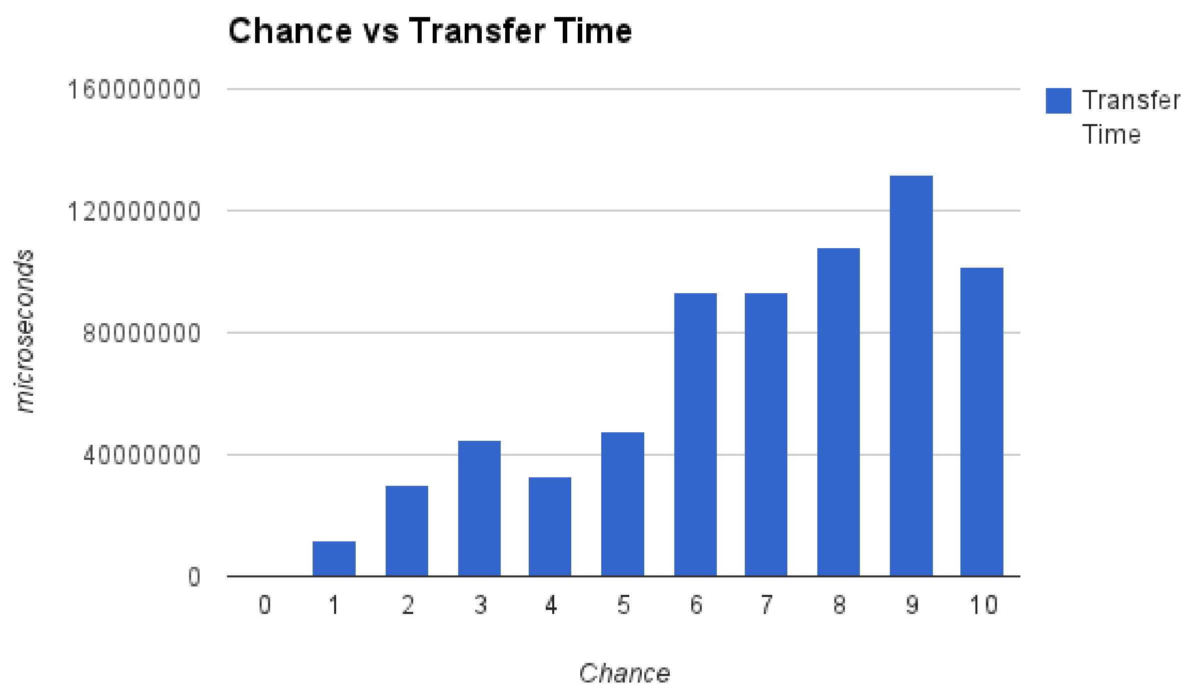
These are similar to the results from the medium sized file, and show an order of magnitude improvement of Go-Back-N over Stop-and-Wait.



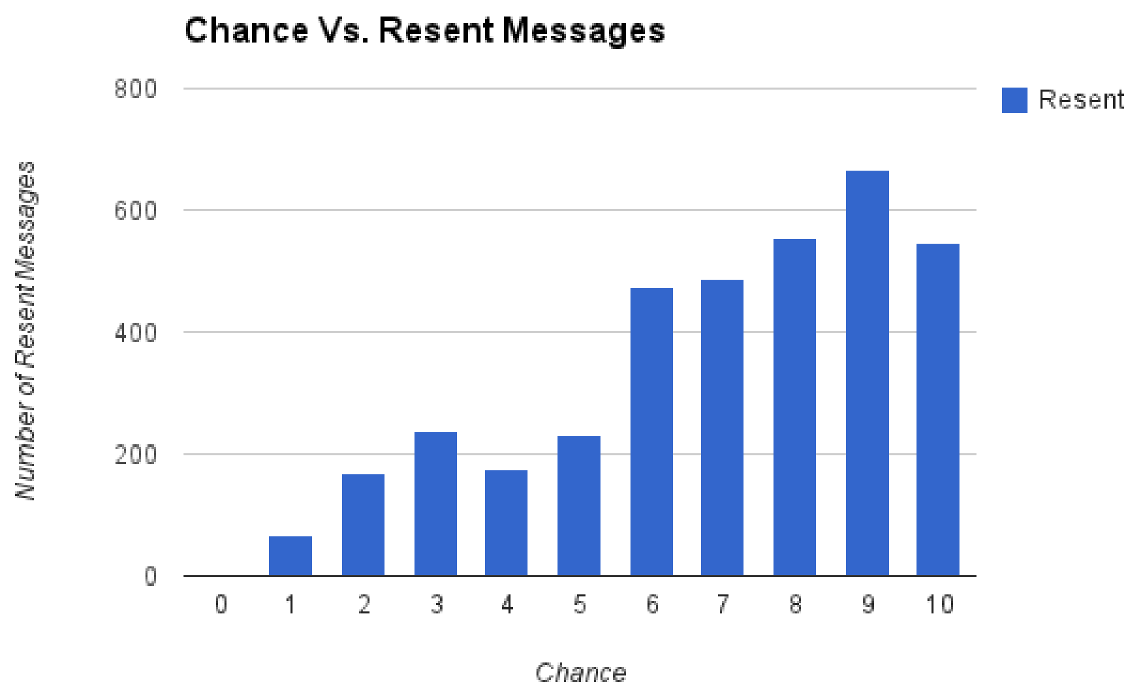
These results are similar to the results from the medium file.

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.



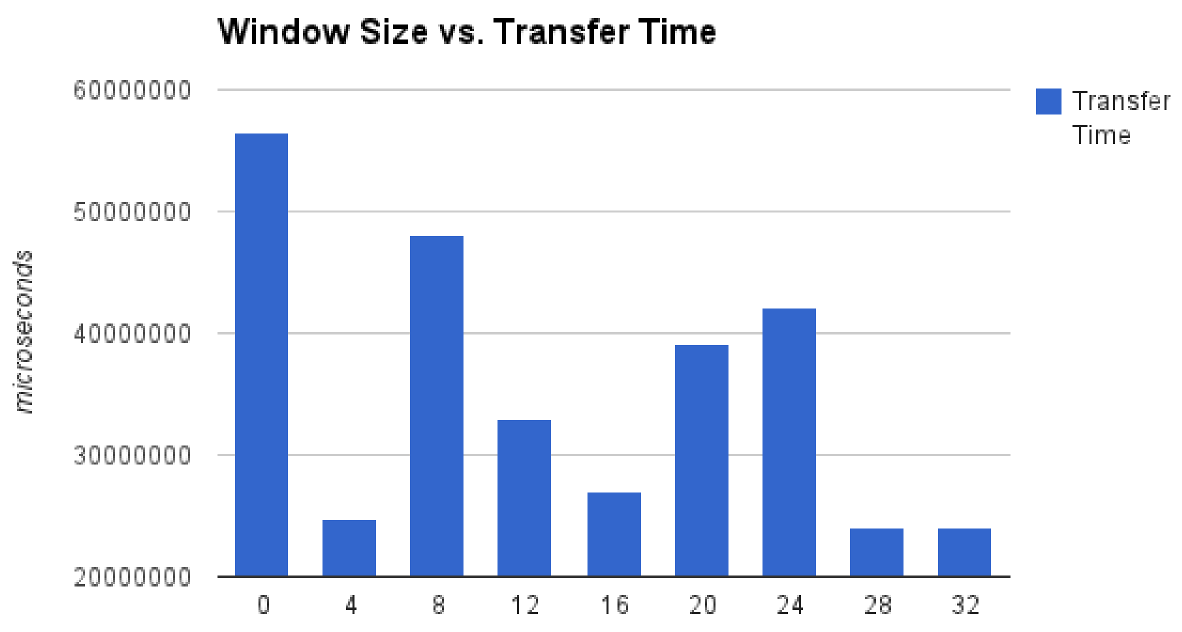
It can be seen that increasing the chance also increases the transfer time. There is a slight anomoly around values of N of 3 and 9, but this is likely just the randomness of the chances creeping in. I would expect with larger files or more trials that there would be a steadier increase in transfer times.



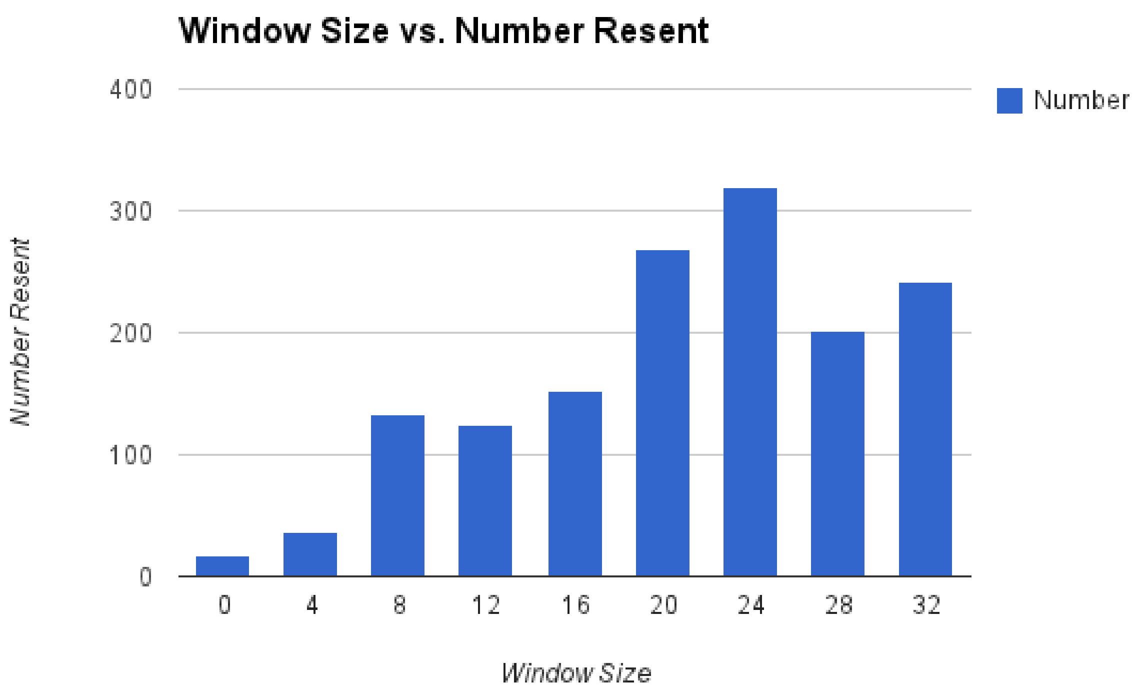
It can be seen that this graph closely mirrors the transfer time. This is not surprising.

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 my program 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:



This graph is a bit confusing. There is a general downward trend, but it is not clear what the best value is. The only thing that is clear is that a window size of 0 takes longer than any of ther other window sizes.



This graph is a bit less confusing, and there is an obvious increase in the number of resent packets. However, the graph peaks at around 24 packets, which is contrary to the expectation that a higher value for the window size would result in a higher number of resent messages, because a dropped, mangled, or delayed packet would result in more packets being resent with Go-Back-N.

Summary

It is clear from these results that Go-Back-N utilizes available bandwidth much more effectively than Stop-and-Wait. The only instances when Stop-and-Wait performs close to Go-Back-N is when the file are small in size. It might also be useful in situations when bandwidth must be conserved or there is a high amount of packet loss. In all other situations, Go-Back-N is a better choice.