Implementation and Testing of a Network Buffer Discovery Application

Jason Anderson

## Implementation - Client

My client implementation is a simple message generator. It allows the specified arguments, plus optional arguments for specifying payload size and burst length. If no burst length is given, it defaults to beginning with 200, and doubling the length with each burst. My method of manipulating the beginning 20 bytes of each message is with a header struct pointer to the buffer to be sent.

## Implementation - Server

I wanted my server to allow for packet reordering, so I decided to keep data on all bursts received for later analysis, rather than trying to detect the end of a burst for data analysis. To do this, I implemented a simple generic dynamic array, found in vector.h. The server keeps a vector called burst_data_vector that contains burst_data for each incoming burst. The burst_data struct in turn keeps a vector of size burst_len that records data about each message. As each message is received, the appropriate burst_data is updated, including keeping timestamps for each arrival. I felt that postponing the analysis until all packets had been received would allow for more flexibility.

When an interrupt is received, the analysis of each burst begins. Received messages are counted and compared to the burst_len to calculate loss rate. My reasoning for determining the buffer size is simple: the first dropped packet of the burst indicates the size of the buffer. The burst analysis will locate the first missing sequence number (message **k**) in the burst, and assume the previous packets filled the bottleneck buffer, giving a buffer size of **k** messages (0-indexed).

Since the server also records timestamps for all received packets, a simple difference between the timestamps of message **0** and message **k-1** gives the time spent to process the buffer. The data size of the messages is calculated with the assumption that ethernet packets are limited to 1472 byte payloads with 42 byte headers. Bytes on the wire are calculated as:

[ceiling(len(message) / 1472) * 42 + len(message)] * k

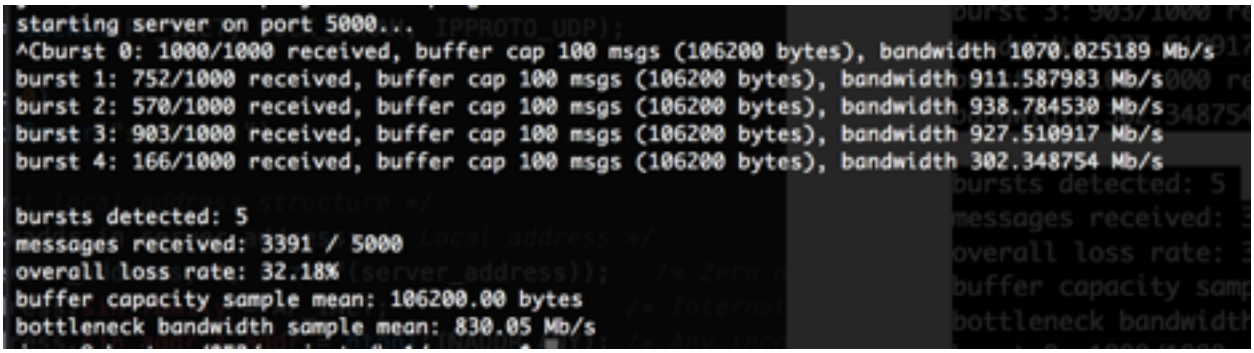
which, divided by the time difference between message **0** and message **k-1**, gives the bandwidth in bytes per second.

To calculate the sample mean of the random variable, I implemented a numerically stable method of calculating mean and standard deviation suggested by [1], found in stats.h. Each burst's tested buffer size and bottleneck bandwidth is averaged, and finally output in the results.

Finally, to implement the special test case with a simulated buffer of size **k**, I simply made the analysis portion of the program ignore packets of sequence number greater than **k**, which achieved the desired results for testing.

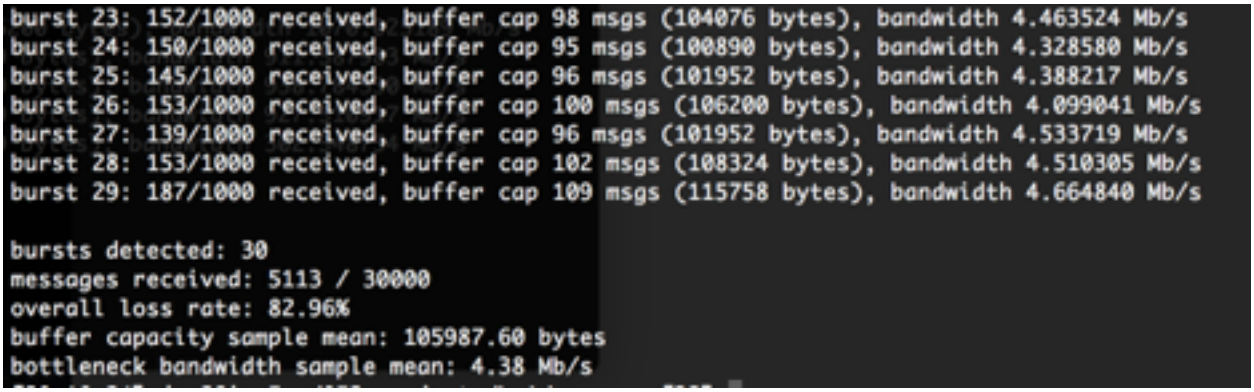
**Test Procedure 1**

I ran both the client and the server in the Ubuntu 14.04.1 VM, using a simulated buffer of 100 messages as described above. The client sent 5 bursts of 1000 messages with 1 second delays between bursts, each with a 1000 byte payload in addition to the 20 byte header. I found that the results were highly variable, but did indeed find indicate a 100 message buffer.

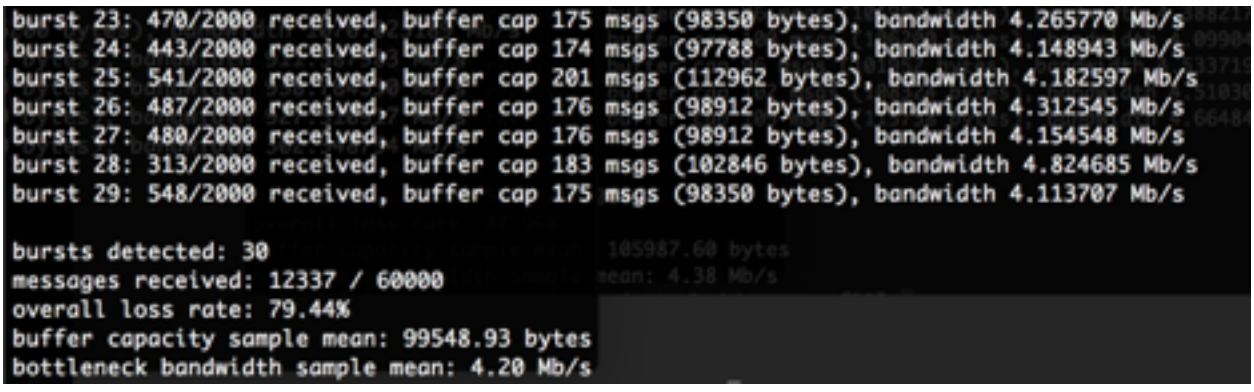



**Test Procedure 2**

I next compiled and ran the server program on one of the ‘imp’ machines in the CS lab at Clemson, without a simulated buffer. My client program was running on the VM on my laptop, which was connected through VPN to Clemson from my home, using Charter cable internet. I tested my average latency to be ~15ms. I again used bursts of 1000 messages with 1000 byte payloads, this time using 30 bursts with 1 second between them:




These results were much more reliable, but I didn’t know if the bottleneck buffer was set to 100 IP messages or ~106KB. I conducted another test, this time with bursts of 2000 messages with 500 byte payloads:



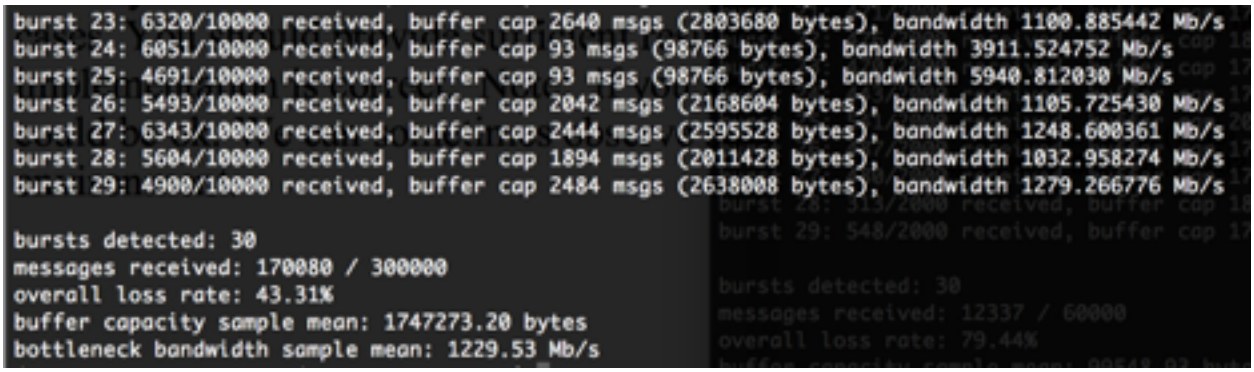
These results seemed to indicate that the buffer was indeed limited by size and not message pointers.

**Test Procedure 3**

I again ran the server on my VM, and this time used tc to adjust the input buffer, delay, and loss rate with the command:

tc qdisc add dev eth0 root netem limit 100 delay 30ms loss 3%

I found that to get more consistent filled buffers with each burst, I needed to send bursts of 10000 messages with payloads of 1000 bytes. My results are as follows:




I found that the variation was still very large, and seems to ignore the 100 message output buffer specified by tc, which leads me to believe that the messages are being processed faster than the output buffer can be filled.

**Conclusion**

After testing, I am confident that my approach to determining the size of a bottleneck buffer is correct, at least for the drop-tail type of buffer management algorithm. My tests probing a real buffer in procedure 2 were consistent, and indicated a ~100KB buffer somewhere along the path.

[1] Donald E. Knuth (1998). The Art of Computer Programming, volume 2: Seminumerical Algorithms, 3rd edn., p. 232. Boston: Addison-Wesley.