Storm Gaming:

A different approach to cloud gaming

by Mark Pierce

Introduction:

Current trends have moved towards trying to utilize the internet and the cloud to create all new kinds of systems and products. This has created a whole new set of problems for network programmers and developers. Such problems are apparent in all systems which try and use any sort of cloud computing. One such system is called cloud gaming, which allows users to play video games without having the game installed on the client's machine. The current systems either use a custom streaming program or a browser based system to play their games. The project outlined in this report is a different approach to creating a cloud gaming system.

Related work

[1] https://cs.uwaterloo.ca/~bernard/edgecloud

Currently implemented cloud gaming systems:

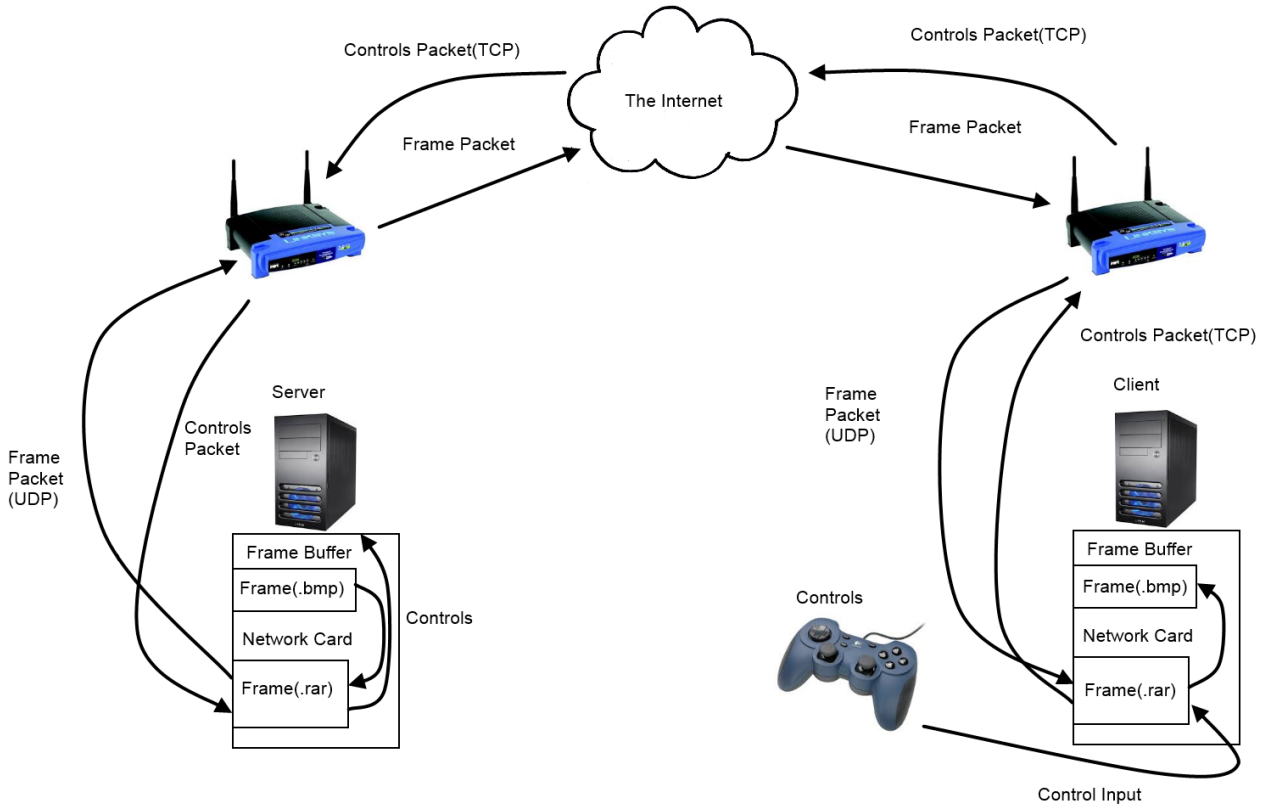
Onlive: http://www.onlive.com/

Gaikai: http://www.gaikai.com/

What Is Storm?

Storm Gaming is the implementation of a new approach to gaming over the internet. The purpose behind its design is to remove as much hardware dependency from the client as possible. Instead of sending encoded video over the internet, the video frames themselves are sent from the server over the internet and put directly into the frame buffer of the client machine. Streaming video this way removes all graphical stress from the client side hardware, as all the graphical calculations and rendering are done on the server side hardware where the game is running. In the current iteration of Storm, the frames are sent as UDP packets in order to minimize the delay between sending the frames from the server and receiving them on the clients side. The framework of Storm can been seen in the visualization figure. 1.

Figure 1. Visualization of Storm concept



What Does this project set out to achieve?

When work on Storm began, it was never meant to be a functioning cloud gaming service. Instead, the purpose of Storm is to test the limits of the current internet architecture to find the challenges and bottlenecks facing network system designers and programmers. The current version of Storm is a mostly theory oriented test of the video streaming method that Storm suggests. This project focuses on the performance achieved when sending and receiving the video frames and comparing that to what would be required to make Storm a feasible system. The other parts of cloud gaming such as handling audio and controls are not within the scope of this project. This is a highly optimistic system that is not necessarily feasible on the current systems. The theoretical tests for this system are as follows.

Uncompressed frames:

1 frame = 1.83MB:

Very poor performance: 10fps: 10fps x 1.83MB = 18.3MB/s

Playable performance: 30fps: 30fps x 1.83MB = 54.9MB/s

Ideal performance: 60fps: 60fps x 1.83MB = 109.8MB/s

High benchmark testing: 120fps: 120fps x 1.83MB = 219.6MB/s

30fps results become 439.2Mb/s and our 60fps becomes 878.4Mb/s. Already from just these calculations it is apparent that these bandwidth requirements are too high for modern standards, at least for the general public, which have internet speeds of between 2 and 50Mb/s connections. Even the highest end publicly available internet speeds cap out at 1Gb/s which would be running at close to full capacity at 60fps.

Compressed Frames(.rar)

1 frame = 736KB

Very poor performance: 10fps: 10fps x 736KB = 7.36MB/s

Playable performance: 30fps: 30fps x 736KB = 22.08MB/s

Ideal performance: 60fps: 60fps x 736KB = 44.16MB/s

High benchmark testing: 120fps: 120fps x 736KB = 88.32MB/s

This gives a bandwidth of 176.6Mb/s at 30fps and 353.3Mb/s at 60fps. These number are much more feasible for modern internet connection speeds, though are still very high. As well any P2P

applications of Storm would be even more infeasible since the general public has internet upload speeds of between 1 and 10Mb/s which is too slow by a factor of ten. So by these numbers, a traditional client server model seems more appropriate, though Storm has been designed so that the server could be replaced with another peer and then could be used as a P2P system.

How does it achieve this?

The programs written for this project operate under a few assumptions that are not dealt with in the programs. The programs do not get the frames from the frame buffer, all of the data used by Storm is sample data, either constructed for lower limit testing, or a sample bitmap image of a fixed size which is compressed. Storm consists of two programs currently, send and receive. These programs were written for the purpose of testing the feasibility of Storm on a single machine, ignoring what internet bandwidth speeds are available currently and just testing the requirements for packing and receiving the frames.

The Data:

Storm has been tested using two different types of data to be sent. The first being a constructed binary text file of the size equivalent to a 800x600x32 bitmap image which could be loaded by the program and sent in packets line by line. This was to simulate an ideal scenario in which the data could be read chunk by chunk and reconstructed easily. The other data that was tested was a sample bitmap image which has been compressed using Winrar to a .rar file. ZIP compression was also tested, but did not compress as much and will not be considered for now, though once real time compression is added, ZIP compression will be tested to see its compression efficiency vs delay loss and how it compares to RAR. The sample image has been compressed to test the benefit of loss-less compression. The limits of this method are that the image is not compressed in real time, and so the delay loss from compression has not been measured. A major goal for future work on Storm is to implement a real-

time loss-less compression of bitmap images which are read from the frame buffer.

Send:

The send program reads the image chunk, whether it be a line from the binary file or a chunk of the compressed RAR file, and loads it into the program, then sends it in a UDP packet to a local port. Then repeats for the next chunk until the frame is sent, then it sends the same frame again, simulating a new frame loaded from the frame buffer. The time is recorded from the beginning of sending packets to when it finished sending the designated number of frames and is outputted to the terminal. This is what is used as the delay between sending and receiving packets, since the receive program does not do anything with the packet as the frame buffer interaction is not implemented yet. The time difference has been compared between measuring when the all the packets have been sent and when all the packets have been received, and the difference is negligible so the former is used for convenience.

Receive:

The receive program listens on the local port and when the port receives a packet, it adds the received packet to its data by appending it to the existing received data, then outputs a acknowledgment to the terminal that it received a packet. In the future, it will write the constructed frames to the frame buffer.

Challenges of UDP:

Storm currently has no implemented solutions for dealing with the drawbacks of UDP, but there are plans for the future to deal with dropped and corrupt packets. The idea for how Storm will deal with dropped packets is straightforward. By using a sequence number of each chunk of a frame, each missing packet will be replaced with the chunk with the same sequence number from the previous packet. This also ties in with how corrupt packets will be dealt with, a checksum will be used to detect corrupt packets and then those packets will be dropped.

Results

The results of running the previously mentioned programs on the given data are shown as follows.

Figure 2. Average Latency vs Frames sent (uncompressed data)

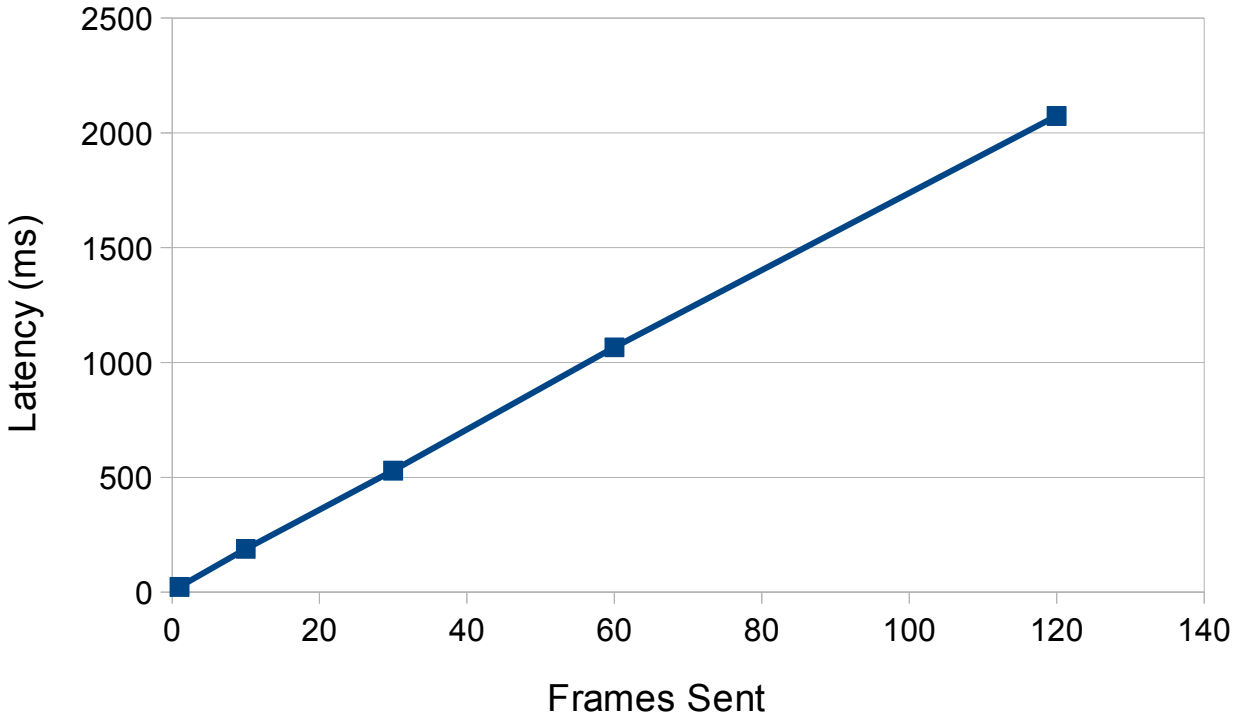
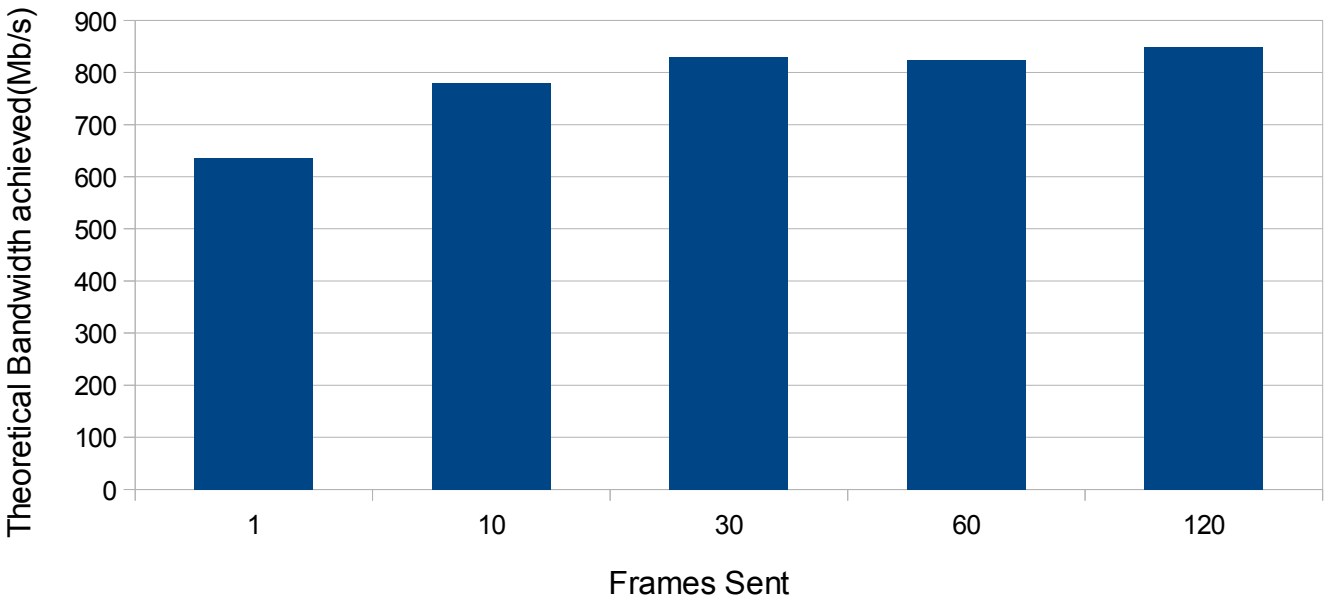


Figure 3. Local Theoretical Bandwidth as calculated from Fig 2.(uncompressed)



The data shown in figures 2 and 3 show that from just packing and sending the uncompressed frame packets, a huge delay is already accumulated. Given that the maximum delay allowed which provides optimal experience for the user is approximately 100ms[1] . Even at 10fps the delay is over 150ms, which is greater than 100ms, so even without restrictions on the internet bandwidth, the current implementation is not feasible.

Figure 4. Average Latency vs Frames sent (compressed)

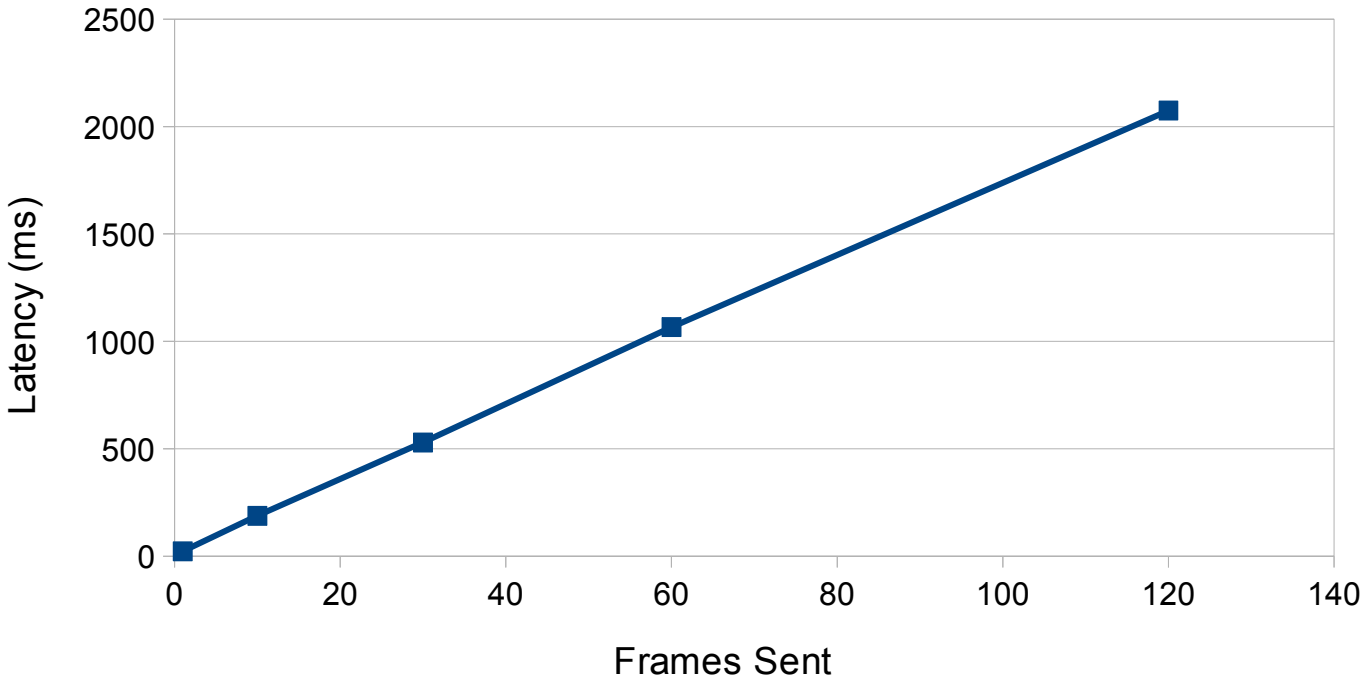
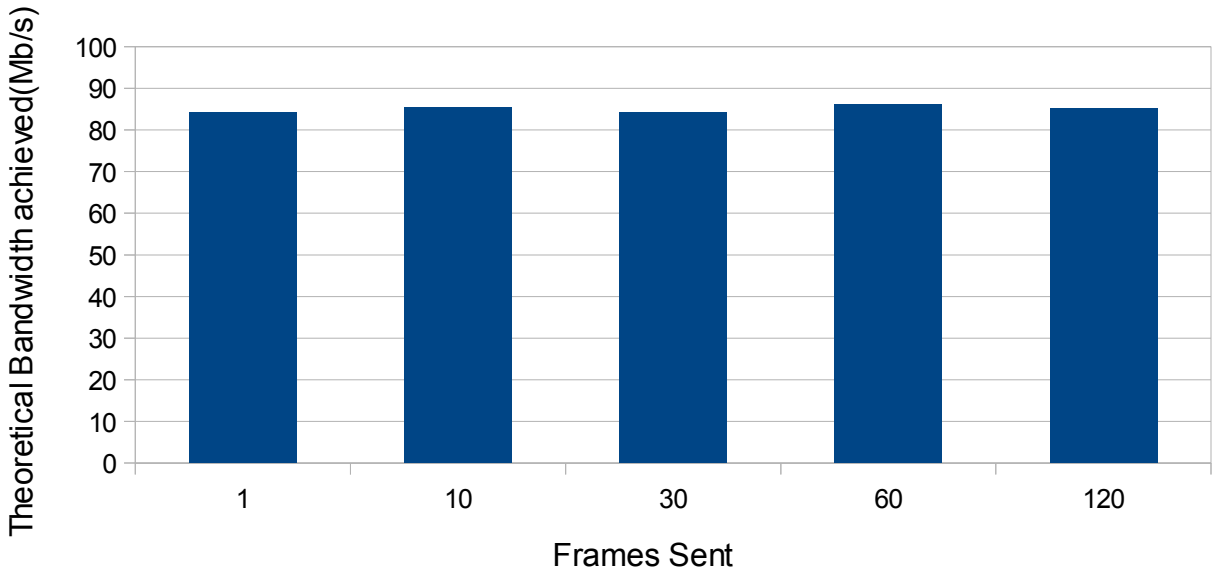
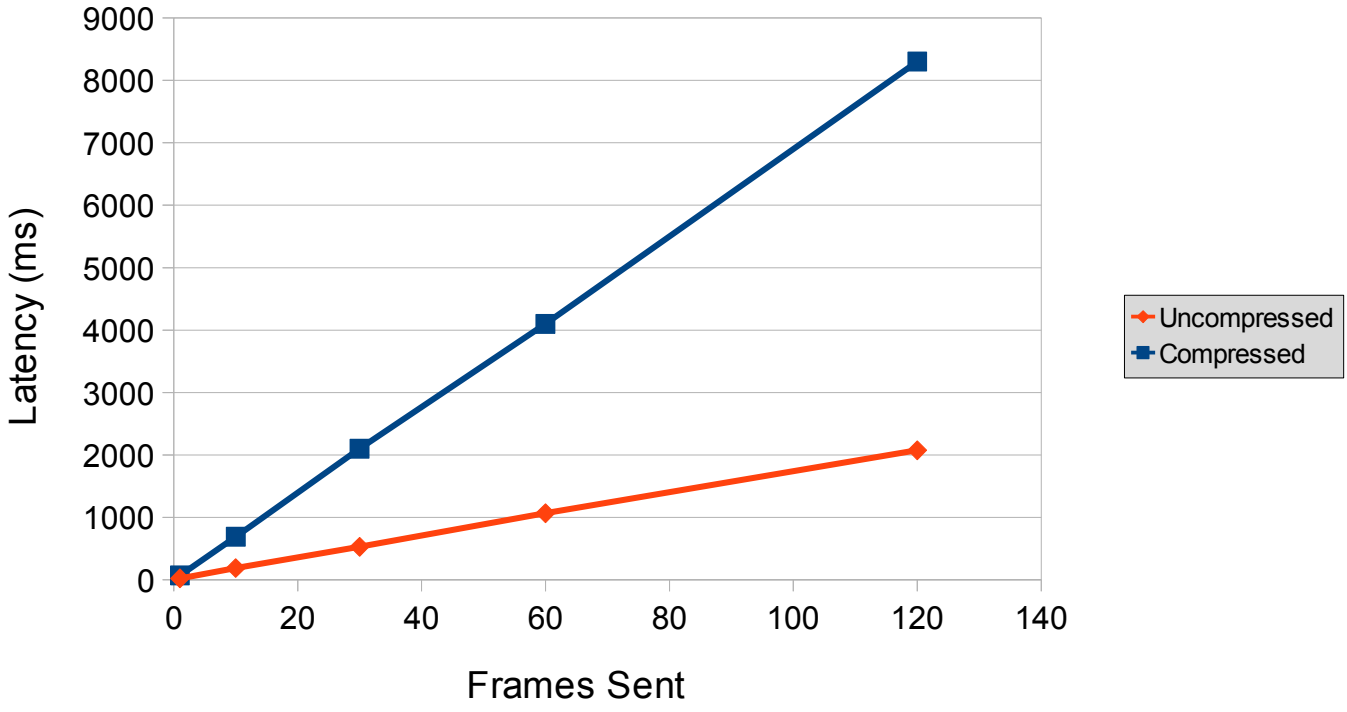


Figure 5. Local Theoretical Bandwidth as calculated from Fig 4.(compressed)



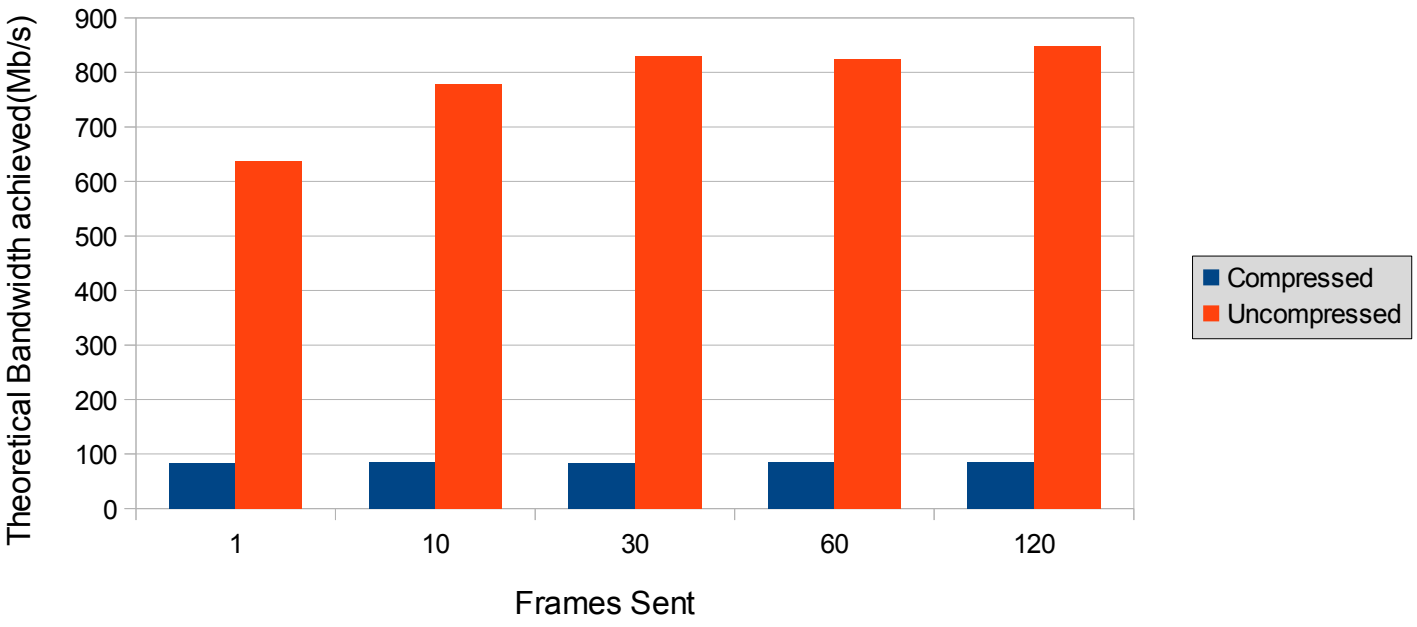
Figures 4 and 5 show that sending the compressed, real bitmap file is much slower than the constructed file. The delay in some cases being over 4 times as high as the constructed file.

Figure 6. Compressed vs Uncompressed Latency



The difference between delays is showcased in figure 6. The uncompressed data delay, shown in orange, is almost four time faster than the compressed data, shown in blue, in all cases.

Figure 7. Bandwidth of Uncompressed data vs Compressed data



The bandwidth achieved by the compressed data is unreasonably low. This is likely due to memory loading delays, but its cause is not known currently known. Future investigation will reveal more. The general trend of testing so far has shown that the delays, not including compression overhead, internet bandwidth delay, packet loss/corruption, are much higher than the target 100ms. The low performance 30fps gives a delay of between 529ms and 2100ms, and high performance 60fps gives a delay of between 1066ms and 4100ms. Though the delays are much higher than the target, previous versions of Storm have had delays of much worse, and through some optimization the delays have been greatly improved, so there is room for improvement on the current iteration of Storm and that should be the main focus of the continuing work on Storm. One solution that would help with memory loading delay in Storm is to have custom hardware, which would allow direct memory access of the frame buffer to a RAR compressor which would store in memory on the network card of the server. This would allow minimal delay for memory access and may cause drastic improvements in delay.

The Future of Storm

I do intended to continue work on Storm, Some ideas for future work on Storm gaming include:

Testing with using TCP

Reliability with UDP

Custom protocols

Run on network simulations and real networks

Real time compression

Frame buffer reading and writing

Conclusion

As the results of testing has shown, Storm is not currently close to being viable with the current internet architecture. It is likely that Storm can be optimized further to increase efficiency and reduce delay, though the current delays of up to 4100ms are magnitudes away from the goal of 100ms. There are many more areas to improve upon in the future of Storm gaming, and new challenges which, when overcome, will bring new improvements to Storm in the future.