Testing the Limitations of System Latency for Virtual Reality

Gabe Terrell
Chris Meyerhoff
Jorge Anton
Oghenefego Ahia
Tomer Shapira
Joshua Howe

Introduction

Scenario

The theoretical scenario of the model revolves around a consumer online-multiplayer VR action game. This exists within the Internet of Things due to many devices being networked together and passing around data in order to create a coherent and playable user experience. This requires significant data processing and event handling because controller and sensor inputs constantly alter the game state. At the same time, each player needs updated data from the cloud and other users many times a second so that a playable environment is maintained.

VR traditionally has high hardware demands. For example, the minimum requirements for the Oculus Rift are[1]:

Graphics Card	GeForce GTX 970 or AMD Radeon R9 290 or better
CPU	Intel Core i5 4590 or greater
RAM	8GB or more
Video Port	HDMI 1.3
USB	2 USB 3.0 ports

As a result, this kind of gaming is not yet accessible to all people; even with recent hardware, VR games may not run smoothly.

Additionally, with any online-multiplayer game, there is a sizeable network requirement; online games have a maximum latency that can be allowed without harming gameplay. With VR, this latency needs to be kept much lower to prevent motion sickness. For a VR game, it is estimated that system latencies above approximately 20 ms will result in a poor experience and potentially motion sickness.[2] As seen in the *Figure 1* below, VR has one of the most stringent latency requirements of any internet application. It requires over ten times less latency than one way video chatting (20 vs 250ms) and even five times less latency than cloud assisted car driving (20 vs 100ms). Therefore, communication between a player's computer and the server needs to be as fast and efficient as possible. Likewise, figure 4 at the end of the document gives an example scenario that shows the importance of low latency in resolving events that happen in online games, and the same concept can be extended to motion within a VR game. VR gaming has not truly taken off yet due to the high system hardware requirements, the price of VR gear, and the inconsistent playing experiences. When examining the growing world of VR, the question that is at the forefront of that expansion and the incorporation of new technologies

[1] Binstock, A. (2016). *Powering the Rift*. *Www3.oculus.com*. Retrieved 17 December 2016, from https://www3.oculus.com/en-us/blog/powering-the-rift/

[2] Qualcomm. Making Immersive Virtual Reality Possible in Mobile. Online. March 2016.

is one of latency. That is what we proposed to explore: what is the current achievable latency for a VR multiplayer game?

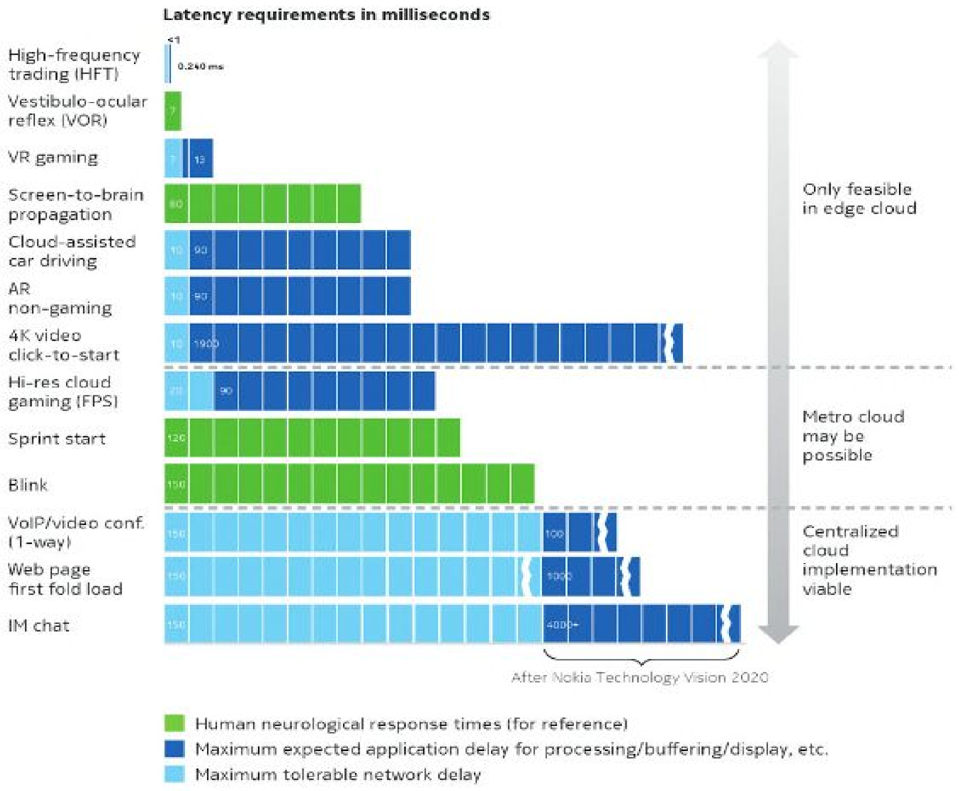


Figure 1: This graphic shows the application latency requirements compared to neurological response times[3]

Sources of Latency

Since everyone does not share the same exact hardware and network infrastructure, the latencies across systems will not be consistent. In order to develop our modelling approach, we first needed to identify the various sources of latency that would have to be incorporated into our model. Beginning with the device, which for our purpose encapsulates the user's computer and VR hardware up to the network, the sources of latency identified would be processing sensor input, updating the game state, and rendering and updating frames. However, the time required to render and update frames is orders of magnitude larger and is strongly impacted by each user's hardware. Therefore, where hardware is not the same user to user device latency will differ.

Likewise, network speed is not constant across all users. Users can be in different cities or towns and thus have varying network latencies to and from the server. Additionally, because there are varying levels of network congestion, rates of dropped packets could differ based on the time of day or across locations. For the cloud, heavy traffic or congestion could alter how

[3] Weldon, Marcus K. The Future X Network: a Bell Labs perspective. pg. 176. Boca Raton : CRC Press., 2016.

quickly it can respond to requests or even how many requests it can handle without dropping some. Since data consistency across users is extremely important for a playable experience, these problems, that can cause different system and network latencies among users, could pile up and result in a poor playing experiences. Thus, they must be accounted for within our model.

Overview of Modeling Approach

Assumptions

To keep the simulation controlled, several assumptions are made. On the device end, updating the game state is assumed to have negligible latency. At the same time, the controllers and VR headset are hard-wired to the machine, and information sent between them and the player's computer has a constant, but negligible, latency. In addition, we are assuming that the hardware of each player's computer is capable of rendering frames with three different, but constant, latencies depending on each user's hardware tier. With the cloud, we are assuming that it has sufficient memory to buffer incoming requests; once data arrives to the cloud, it is not lost. Our assumptions with the network end of the system are that each unique network route has a constant propagation delay; data sent between Boston and Atlanta would have a constant latency, while the data sent between San Francisco and New York would have a different constant latency (*see Figure 2*). We are also assuming that users can connect with anyone in the country. Lastly, on the gameplay side, processing incoming events has a constant latency cost, and ten people will be playing per game.

Simulator Model

The simulator creates and runs scenarios with different settings, keeping a measure of the overall changing latencies across devices. It creates and manages ten devices in each "game," the single network module, and the cloud server module. During the creation of these, it passes in the different scenario settings. Likewise, it manages packet transfer between each of the different modules.

Packet Class/Interface

To provide a consistent interface among modules, there is a common packet implementation that is used to send data between the device, cloud, and network, similar to what would be used in a real deployment of VR. Each packet contains a timestamp, the source ID, the destination ID, and the position of the player in the game. Our model assumes that the player's entire game state can be represented in a single packet of data. For example, the following data structure could be used to represent the player's state at any given point of time:

Event / Stimuli	*Data Representation (excluding timestamp)*
Shooting	Flag Bit
Jogging	Flag Bit
Initiate Reload	Flag Bit
Location (point)	3D Vector (3 Floats: 12 Bytes)
Person Movement Vector	Quaternion Vector (4 Floats: 16 Bytes)
Person View Vector	Quaternion Vector (4 Floats: 16 Bytes)
Bullet Hit	Quaternion Vector (4 Floats: 16 Bytes)
ID	1 Byte

This example would takes less than 75 bytes to represent; so even with additional data parameters, it is a conservative estimate that the player's state can be represented with 75-100 bytes. Assuming a ten player game, the state of the entire game can fit into a packet under 1000 bytes, which is well within the range of an average size of a TCP packet.[4] For the simplicity of our own model, we will only send the player's x-y position to the cloud, but assume the above packet size and that all data can be transmitted within one packet.

Device

Each device is assumed to send a constant number of events and updates to the cloud per second to accurately model a modern multiplayer game. Likewise, each device exists in one of three hardware tiers that is capable of maintaining a constant fps; those tiers are 120, 90, and 60 fps. The simulated VR headsets which correspond to each tier are as follows:

120 fps	Playstation VR, LG 360 VR
90 fps	Oculus Rift, HTC Vive
60 fps	Gear VR, Visus VR

These tiers are modeled by adding a constant amount of latency, either 8.3 ms, 11.1 ms, or 16.6 ms (corresponds to 120, 90, and 60 fps respectively), to the overall latency measurement.

[4] "Packet Size Distribution Comparison Between Internet Links in 1998 and 2008." Center for Applied Internet Data Analysis. University of California San Diego, Web. 18 Dec. 2016.

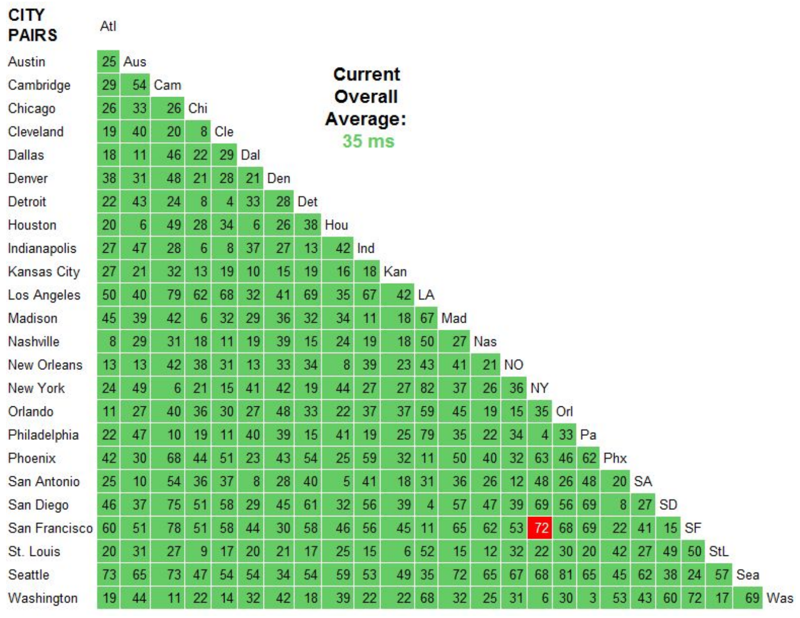
Every device has a location and a unique ID, which allows for it to be identified by the cloud and network. During creation, each device is passed its hardware tier, location, and the number of events/updates per second to be sent to the cloud. Each device also keeps a measure of its own latency and logs that data over the course of a game.

Network

The network model receives and forwards packets from the device and cloud, processes dropped or lost packets, and simulates either TCP/UDP. These translate into latency as propagation delay, packet loss and recovery, and protocol overhead. The propagation delay is constant between two edge nodes based on the aforementioned assumptions. Specifically, the numbers from *Figure 2* below are used to increment the overall latency. Packet loss is set as a probability per simulation, which could vary based on network congestion rates which also affect the cloud. The protocol overhead and packet loss varies by protocol.

The TCP Network model utilizes an acknowledgement model to insure packet reception. When packet loss occurs, the TCP protocol will trigger a timeout (due to lack of an acknowledgement) and will cause the sender to resend its data. The TCP protocol overhead is larger than UDP, and the need to keep the socket in use to listen for an acknowledgement from the receiver causes the overall protocol to perform slower than UDP.

The UDP protocol utilizes an datagram sending model that offers no guarantee of packet loss. If a packet is lost, the cloud will continue sending fresh data regardless of the information being lost and accepts that the players will incur a performance decrease from jumpy data. The protocol overhead is simpler than TCP.



CITY PAIRS	Atl	Aus	Cam	Chi	Cle	Dal	Den	Det	Hou	Ind	Kan	LA	Mad	Nas	NO	NY	Orl	Pa	Phx	SA	SD	SF	StL	Sea
Austin	25																							
Cambridge	29	54																						
Chicago	26	33	26																					
Cleveland	19	40	20	8																				
Dallas	18	11	46	22	29																			
Denver	38	31	48	21	28	21																		
Detroit	22	43	24	8	4	33	28																	
Houston	20	6	49	28	34	6	26	38																
Indianapolis	27	47	28	6	8	37	27	13	42															
Kansas City	27	21	32	13	19	10	15	19	16	18														
Los Angeles	50	40	79	62	68	32	41	69	35	67	42													
Madison	45	39	42	6	32	29	36	32	34	11	18	67												
Nashville	8	29	31	18	11	19	39	15	24	19	18	50	27											
New Orleans	13	13	42	38	31	13	33	34	8	39	23	43	41	21										
New York	24	49	6	21	15	41	42	19	44	27	27	82	37	26	36									
Orlando	11	27	40	36	30	27	48	33	22	37	37	59	45	19	15	35								
Philadelphia	22	47	10	19	11	40	39	15	41	19	25	79	35	22	34	4	33							
Phoenix	42	30	68	44	51	23	43	54	25	59	32	11	50	40	32	63	46	62						
San Antonio	25	10	54	36	37	8	28	40	5	41	18	31	36	26	12	48	26	48	20					
San Diego	46	37	75	51	58	29	45	61	32	56	39	4	57	47	39	69	56	69	8	27				
San Francisco	60	51	78	51	58	44	30	58	46	56	45	11	65	62	53	72	68	69	22	41	15			
St. Louis	20	31	27	9	17	20	21	17	25	15	6	52	15	12	32	22	30	20	42	27	49	50		
Seattle	73	65	73	47	54	54	34	54	59	53	49	35	72	65	67	68	81	65	45	62	38	24	57	
Washington	19	44	11	22	14	32	42	18	39	22	22	68	32	25	31	6	30	3	53	43	60	72	17	69

Current Overall Average: 35 ms

Figure 2: This shows the network latencies in ms between pairs of cities in the U.S[5]*. The thresholds are distance sensitive. These are the constants used in the simulation.*

[5] *Global IP Network Latency*. (2016). *Ipnetwork.bgtmo.ip.att.net*. Retrieved 8 December 2016, from http://ipnetwork.bgtmo.ip.att.net/pws/network_delay.html

Cloud

The cloud is in charge of simulating the latency required for a cloud service, like AWS, to process packets with the updated data for each player. Whenever it receives a packet, it places it in its queue and keeps it there until it is processed. As mentioned previously, the queue is assumed to be of infinite size, so no packet is lost once it reaches the cloud. To more accurately model this, we set the following parameters: game traffic, location of the cloud, timeout, and the number of players. The game traffic can either be high, medium, or low and change the amount of packets the server can process in a certain period of time. The cloud handles the state of every party of the game being played; the more parties, the greater the turnaround time of processing the head of the queue of each party. The location parameter specifies where the cloud servers are located, which affects the propagation delay between each user and itself. Lastly, if the packet has taken more time than the timeout for it to arrive from a user to the cloud, it is ignored and not forwarded to the rest of the users.

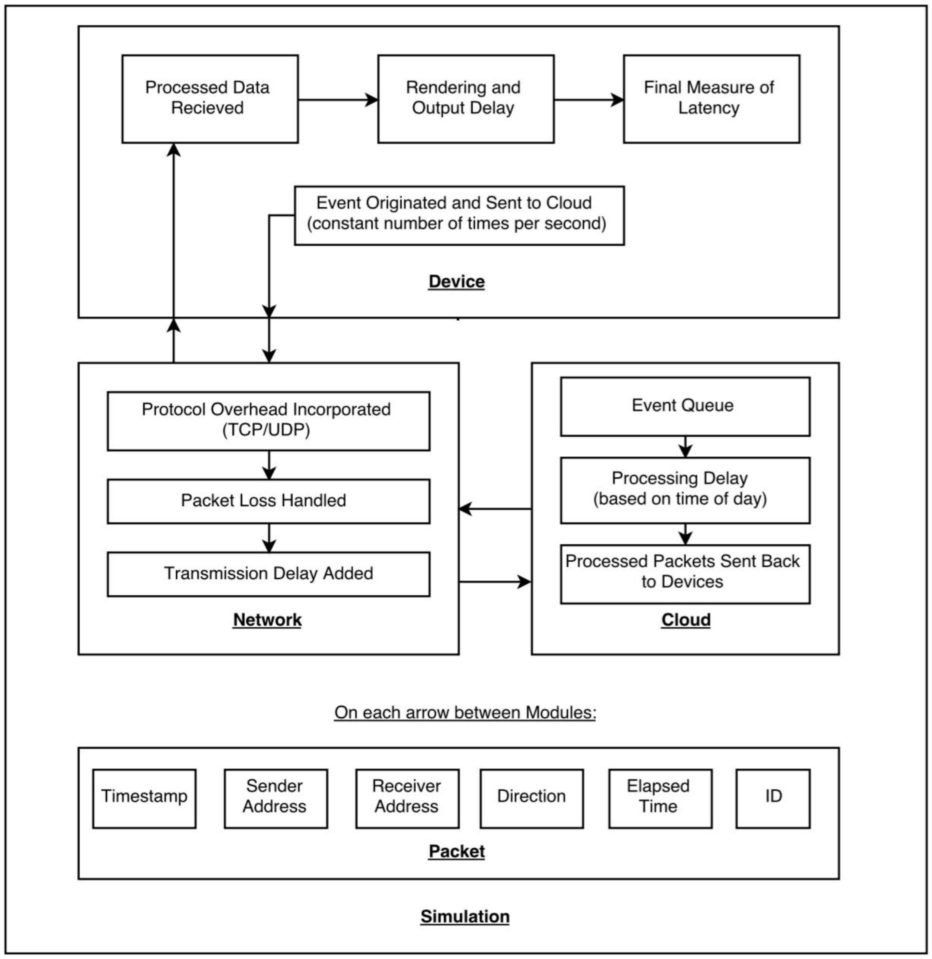

Processed Data Recieved
Rendering and Output Delay
Final Measure of Latency
Event Originated and Sent to Cloud (constant number of times per second)
Device
Protocol Overhead Incorporated (TCP/UDP)
Packet Loss Handled
Transmission Delay Added
Network
Event Queue
Processing Delay (based on time of day)
Processed Packets Sent Back to Devices
Cloud
On each arrow between Modules:
Timestamp
Sender Address
Receiver Address
Direction
Elapsed Time
ID
Packet
Simulation

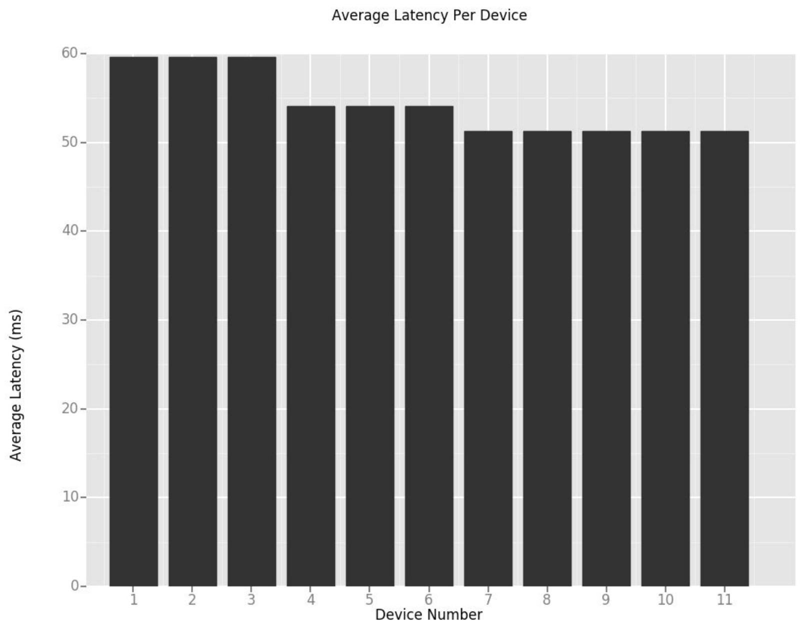
Figure 3: This is a high level view of how the program operates and models latency. Flows between the network, cloud, and devices are performed through the packet interface.

Results and Analysis

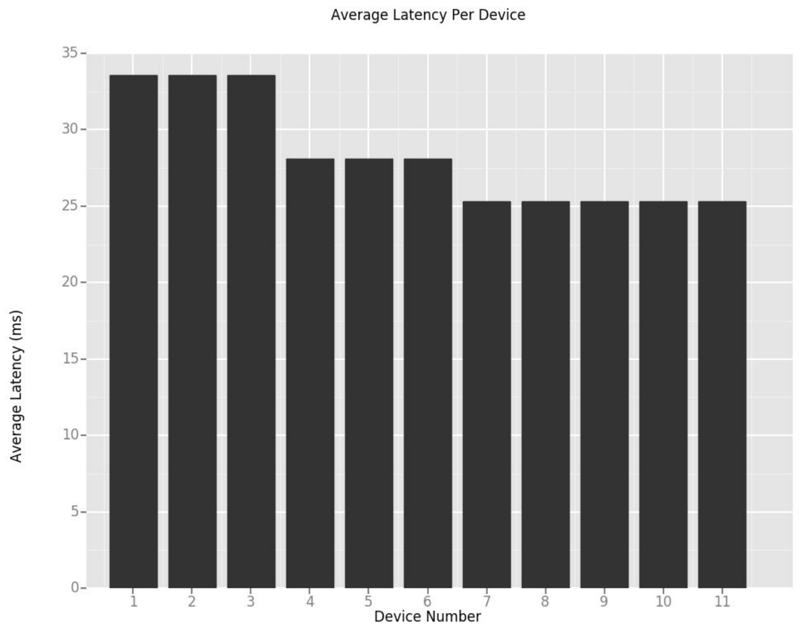
Simulation Data

For each test devices 1-3 are low hardware tier, 4-6 are medium tier, and 7-11 are high tier.

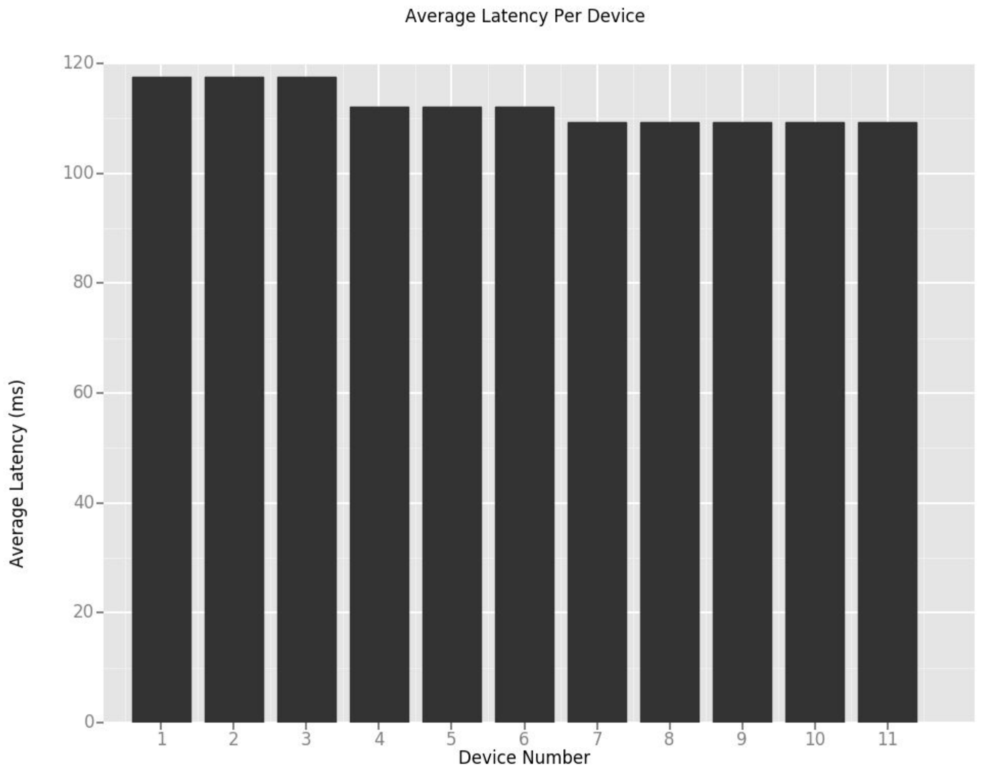
1. TCP and close location, low traffic



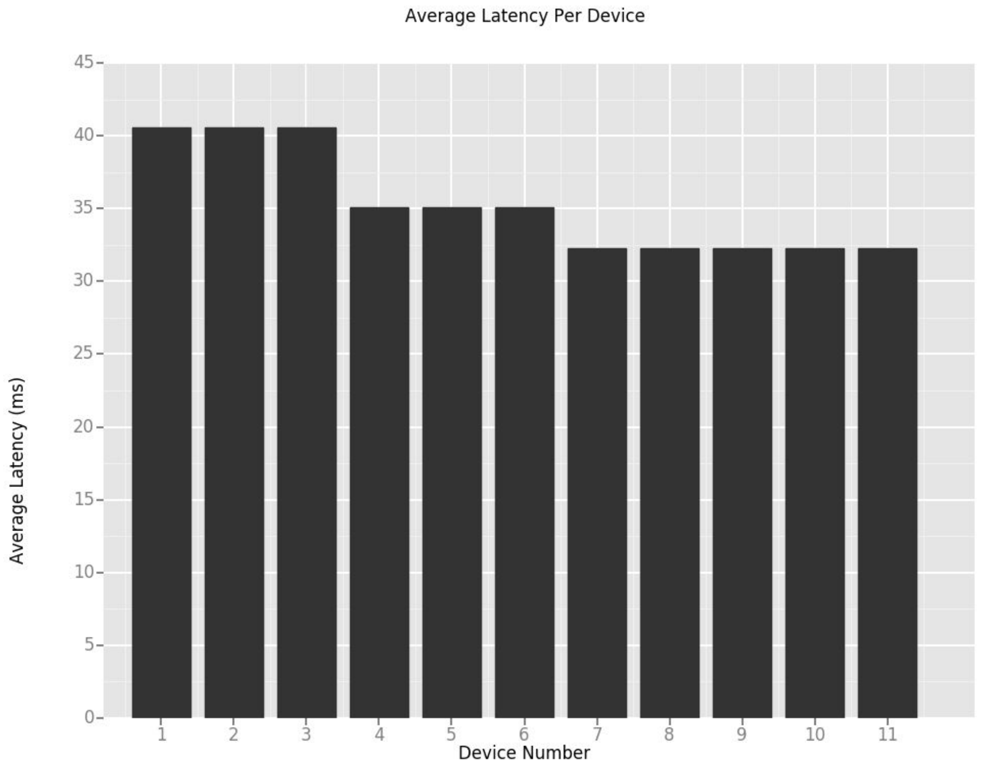
2. UDP and close location, low traffic



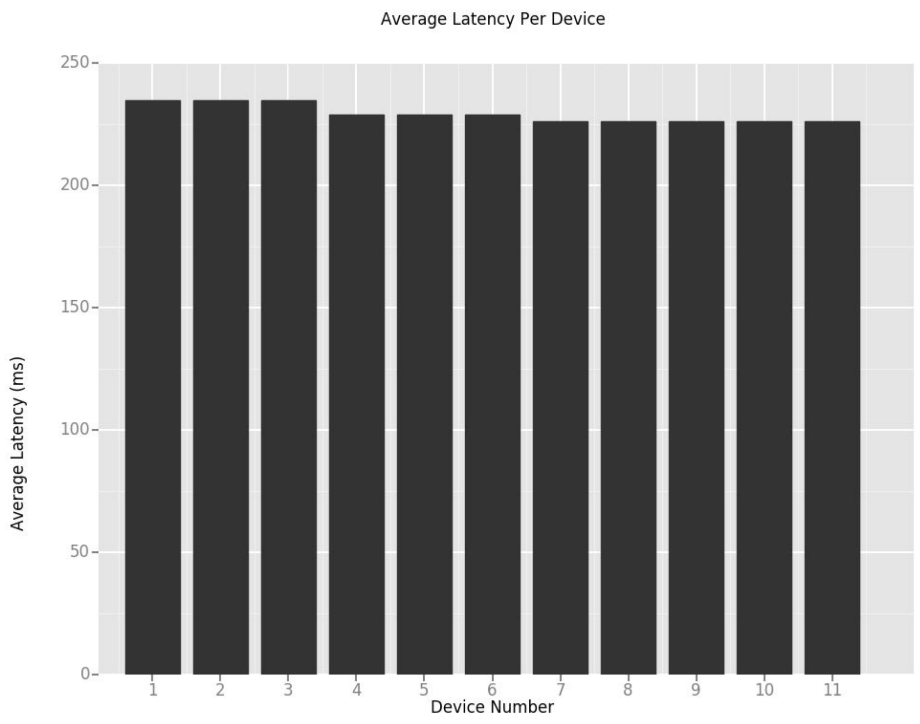
3. Faster protocol of the two and further locations, low traffic



4. Faster protocol, close location, and high traffic



5. Slower protocol, far location, high traffic



Simulation Analysis

The five different scenarios above were tested. Each test involved the three different hardware tiers and their corresponding rendering latencies. The first two tests should determine the best-case scenario: they test both network protocols under the ideal conditions of everyone being in the same city (5 ms of network latency) and low server traffic. The next two tests used the protocol determined to be the fastest (UDP) and tested a single bad condition: either far location or high server traffic. These should have yielded a middle ground in terms of test cases. The worst case test scenario used the slower protocol (TCP), a larger distance between users and the server, and a higher amount of server traffic.

The graph trends generally agree with our theoretical understanding of the scenarios. The first two graphs show the impact of the change of protocol and clearly show that UDP is the better protocol for achieving low latency; with TCP the latencies ranged from 50-60 ms while with UDP they ranged from 25-35 ms. The third graph shows the massive impact that distance has on latency; even with the faster protocol, due to distance (Cambridge to Denver), there is higher latency than there was with the slower protocol with a shorter distance (due to distance latency increased to 110-120 ms). The fourth graph shows the fairly small impact of network traffic (increased latency with UDP by about 7 ms), while the fifth graph shows that in the worst case, a slow protocol and high distance, a latency far above what is considered playable is yielded (225-240 ms). The impact of the different device tiers is more important during the more optimistic scenarios that result in lower latency. In the scenarios with higher distance, the device tiers become a far smaller determinant of overall latency.

One important observation from these results is that games benefit from accepting the performance hit from packet loss. TCP uses acknowledgements to insure that dropped packets make it to the server, but often the cloud in our simulation would end up discarding this data because it arrives so late (from the TCP timeout/acknowledgement combination) that other game devices would not benefit from receiving this data. In the UDP scenarios, a dropped packet is simply gone, and the games will accept that a player may potentially jump a little bit since one of its actions was missed. In either scenario, the dropped packet ends up being discarded (by the router or the cloud), so it makes sense to choose the protocol that operates without reliability in exchange for less overhead. Therefore, UDP is the better protocol for a game where state packets are rapidly being blasted at the server.

After performing all of the tests, we easily saw the importance of having games in which people are matched by their location and with a cloud server that is close to all players. If the cloud was even half-way across the country from all the players, the latencies roughly quadrupled making the game far less playable. The impact of relatively strong hardware, while important for maintaining a complete frame latency low enough to not be sick, only plays a large part of player's information latency when operating under ideal conditions. This is seen in practice too, where a majority of online games have servers in both the West Coast and East Coast, and possibly other servers in the Midwest based on how stringent latency requirements actually are.

Generally the simulation results returned latency values above what is considered an enjoyable VR experience. However, these are the latencies of other players' data and not the frame latencies. What this means is that while most of the frame can be rendered fast enough, the other players actions and movements are coming in too slowly for good gameplay in all cases except the best case/ideal scenario (which is still slightly too high at 25-35 ms). This will result in a potentially laggy experience, which with VR translates to a higher risk for motion sickness.

Future Work

Potential Expansion of Simulation

Future additions to our simulation could include the following:

- A fuller expansion of the expected latency due to the graphics pipeline, and the varying algorithms that can be used to simulate reduced latency (pre-rendering, continuation of other players velocity etc)
    - For example, we could include some of what is talked about here: http://www.gamedonia.com/blog/lag-compensation-techniques-for-multiplayer-games-in-realtime
- The network model could test more algorithms or the same ones in more depth to find out how to minimize latency and how algorithms compare.
- Adding more specific events that simulate realistic gameplay (would entail expanding

what is sent in each packet, and adding specific latency costs)

- Exploring whether it would be possible to offload processing to the cloud, and if so, what latency would occur
- Within the cloud model, the estimation of the processing time could be improved
- Different arrangements of sending updates and events (ie: trying a method where the cloud constantly pushes updates with the most recent data, and players send their updated data in response and when they perform some action)
- Model in-game movement to see how much a player jumps as an alternate measure of latency

---

Works Cited

1. Qualcomm. Making Immersive Virtual Reality Possible in Mobile. Online. March 2016.
2. Weldon, Marcus K. The Future X Network : a Bell Labs perspective. pg. 176. Boca Raton : CRC Press., 2016.
3. Liu, Xian. Deterministic Latency Network Use Cases. The Internet Engineering Task Force (IETF®) Network Working Group. October 31, 2016
4. "Packet Size Distribution Comparison Between Internet Links in 1998 and 2008." Center for Applied Internet Data Analysis. University of California San Diego, Web. 18 Dec. 2016.
5. Binstock, A. (2016). *Powering the Rift*. *Www3.oculus.com*. Retrieved 17 December 2016, from https://www3.oculus.com/en-us/blog/powering-the-rift/
6. Global IP Network Latency. (2016). Ipnetwork.bgtmo.ip.att.net. Retrieved 8 December 2016, from http://ipnetwork.bgtmo.ip.att.net/pws/network_delay.html
7. Xicota, David. "Lag Compensation Techniques for Multiplayer Games in Realtime." Gamedonia. 29 Feb. 2016. Web. 19 Dec. 2016.

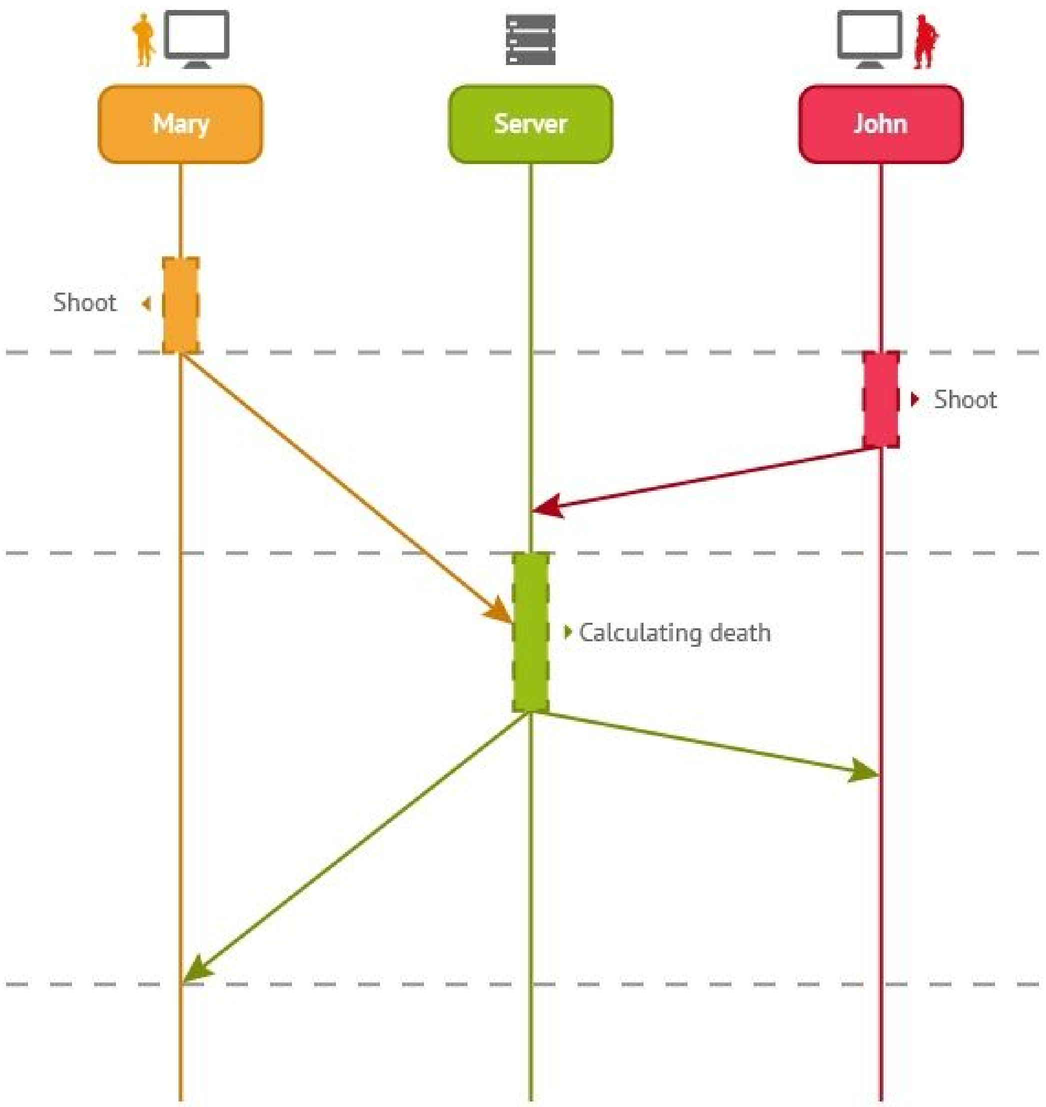


Figure 4. Shows the effects of latency on determining what happens in gameplay. Even though John shoots after Mary does, his packet/action reaches the server before Mary's does, and thus the server calculates that Mary dies rather than John.

Source:
Xicota, David. "Lag Compensation Techniques for Multiplayer Games in Realtime." Gamedonia. 29 Feb. 2016. Web. 19 Dec. 2016.