IMPROVING VIDEO TRANSPORTATION OVER DYNAMIC WIRELESS NETWORKS

by

Joshua B. Groen

A thesis submitted in partial fulfillment of
the requirements for the degree of

Masters of Science

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2017

Thesis approved by:
Xinyu Zhang, Assistant Professor, Electrical and Computer Engineering

To my family, for their constant support.

ACKNOWLEDGMENTS

I am indebted to many people for their invaluable help in getting this far. I have known some great leaders in the Army who pushed me to achieve and enabled me to take the opportunity to complete my Master's Degree including: LTC Jim Caryl, LTC Kirk McIntosh, LTC Joe Jasper, and MAJ Jason Kendzierski. Gentlemen, thank you. To my professors at UW Madison, who pushed me to learn, I am grateful. Especially Xinyu Zhang, who consistently encouraged me to go beyond my comfort zone; thank you. I must thank my family for their love and support. My three sons are a great joy to me and a constant source of motivation: Ryder, Gavin, and Wyatt. Boys I love you. My wife Autumn has consistently held down the fort, when I was training, deployed, or hiding in the basement working on homework or this Thesis. Thank you babe.

— Joshua B. Groen (2017)

CONTENTS

Contents iii

List of Tables iv

List of Figures v

Abstract vii

1 Introduction 1

2 Improving Transport Layer for Bulk Video Transfer 4
- *2.1 Background* 4
- *2.2 Relationship Between Throughput and Delay* 13
- *2.3 Design of Congestion Control Algorithm* 16
- *2.4 Experimental Results* 21
- *2.5 Further Analysis* 25
- *2.6 Backward Compatibility* 28

3 Improving Virtual Reality Video Streaming 30
- *3.1 Background* 30
- *3.2 Improving VR Streaming Through Video Processing* 31
- *3.3 Improving VR Performance with Edge Computing* 43
- *3.4 Practical Implementation* 48

4 Conclusion and Future Work 51

References 53

LIST OF TABLES

2.1 Impact of Competing Flows on rtt. 14

3.1 Comparison of CPU Utilization for different tile schemes. 35
3.2 Comparison of bandwidth utilization for different tile schemes based on 5 different videos. 43

LIST OF FIGURES

1.1 Internet Access 2000 to 2015[15]. 1

2.1 Throughput and Delay for LTE Network. 4
2.2 Standard TCP Cubic (L), Modified TCP Cubic (R). 13
2.3 Impact of Competing Flows on Congestion Window Size. 14
2.4 Congestion Window Size Impact on rtt. 15
2.5 Congestion Window Without and With Delay Count. 21
2.6 TCP Cubic: Congestion Window and Throughput vs Delay. 22
2.7 TCP Vegas: Congestion Window and Throughput vs Delay. 22
2.8 Verus: Throughput and Delay. 23
2.9 My Algorithm: Congestion Window and Throughput vs Delay. 23
2.10 TCP Cubic with Two Users: Congestion Window and Throughput vs Delay. 24
2.11 My Algorithm with Two Users: Congestion Window and Throughput vs Delay. 24
2.12 TCP Vegas with Two Users: Congestion Window and Throughput vs Delay. 26
2.13 Verus with Two Users: Congestion Window and Delay. 26

3.1 CPU utilization with no tiles. 34
3.2 CPU utilization for DASH with SRD. 34
3.3 CPU utilization for 9 high definition tiles. 35
3.4 CPU utilization for 5 high and 4 low definition tiles. 35
3.5 Spherical Coordinates. 36
3.6 Spherical to Equirectangular Mapping. 37
3.7 Average pdf for several movies. 38
3.8 Option 1. 39
3.9 Option 1, Number of Tiles Required. 40
3.10 Option 2. 41

3.11 Option 2, Number of Tiles Required. 41
3.12 Option 3. 42
3.13 Option 3a, Number of Tiles Required for Equitorial Set. 42
3.14 Option 3b, Number of Tiles Required for Polar set. 42
3.15 Round Trip Time to Wi-Fi AP vs Congestion. 45
3.16 Run Times for 4K Video. 46
3.17 Run Times for HD Video. 46
3.18 Run Times with Pause. 47
3.19 Run Times with Pause and Skip Forward. 47

ABSTRACT

Improving Video Transportation over Dynamic Wireless Networks

This thesis explores video transfer over dynamic wireless communication links in an attempt to understand current limitations and propose new methods to push performance beyond the existing envelope. To this end, I spent considerable time researching and experimenting with two wireless communication methods, 4G LTE and Wi-Fi. I also looked at different problem sets that currently limit these specific technologies. This thesis is therefore broken down into three main sections.

The first major section explores the performance of TCP over live 4G LTE networks in order to gain an understanding of current congestion control protocols and help guide the design of a new congestion control algorithm. Specifically, this section of the thesis attempts to determine the relationship between throughput and delay over LTE using different existing TCP congestion control algorithms. It then explores the possibility of designing a new delay based congestion control algorithm based on the best features of current congestion control algorithms observed in practice. This sections details the proposed algorithm which is built on top of the existing TCP Cubic congestion control algorithm in the Linux Kernel and tested over a commercial LTE network.

The second major section focuses on improving the performance of Virtual Reality (VR) systems. Streaming VR videos is extremely bandwidth intensive, and also highly sensitive to delay. This highlights the key features of throughput and delay and illustrates the necessity of understanding how to achieve the best performance. This section details the current performance of VR systems and proposes innovative ways to improve performance. These methods focus on two possibilities; reducing the data to be sent through tiling the video and use of DASH, and shifting some of the core functions from a distant server to the very edge of the wireless network - the wireless access point (AP).

1 INTRODUCTION

Our lives are both more connected and more mobile than ever before. Globally in 2015, 3.2 billion people are using the Internet [15]. This represents a tremendous growth in the use of the Internet. But the growth rate in mobile devices is even greater. In 2015 there were more than 7 billion mobile cellular subscriptions world wide representing more than 700 percent growth in the past 15 years [15]. This can clearly be seen in Figure 1.1 which graphs the growth of connectivity from 2000 to 2015. The two curves with the highest growth over the past 15 years are the Mobile-cellular telephone subscriptions and the Mobile broadband subscriptions. This means that more than ever before the last hop of our on-line interactions is wireless. In fact, Cisco predicts that by 2020 wirless and mobile devices will account for two-thirds of internet traffic [5].

The bulk of current internet traffic is video traffic. Cisco reports that in 2015, video traffic accounted for 70% of all Internet traffic and they predict that it will account for 82% of all Internet traffic by 2020 [5]. By 2020, nearly a million minutes of video content will cross the network every second [5]. A notable subsegment of the video traffic is virtual reality (VR) traffic. VR traffic alone quadrupled from

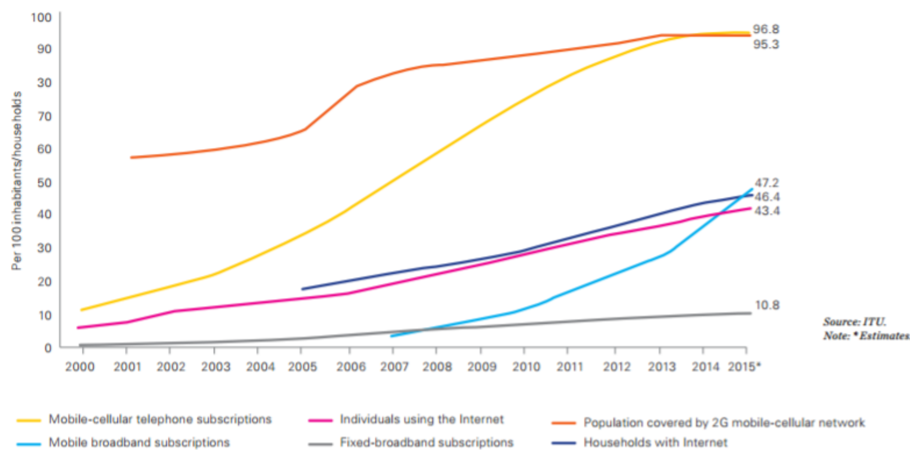

Figure 1.1: Internet Access 2000 to 2015[15].

2014 to 2015, and Cisco predicts it will increase 61-fold from 2015 to 2020 [5].

Optimizing throughput over the Internet is not a new problem. It has been studied for many years, with many previous works including [26, 19, 48, 7, 9, 11, 37, 3, 42, 16, 33, 8, 23, 43, 10]. While this is an old problem in general, there are new aspects that greatly change the framework in which we must work to solve these problems. First, the incredible growth of cellular networks means we must focus more attention there. Cellular networks are fundamentally different from traditional wired networks. Both the channel capacity and the traffic load are far more variable than even Wi-Fi networks. This presents many problems that must be addressed. The second new aspect is the huge growth in VR traffic. Streaming videos already requires high bandwidth and low latency. Because VR videos consist of an entire 360° degree field of view and contain a separate video stream for each eye they have far higher bandwidth requirements still.

This thesis specifically looks to solve two main issues related to video transportation. The first issue that is covered is optimizing the transport layer for bulk video transmission over dynamic wireless links. Specifically, Chapter 2 evaluates the current TCP layer performance of LTE and proposes solutions to optimize TCP performance over LTE. In this section I conducted many experiments over live, commercial LTE networks. LTE is, in general, a mature technology with many good papers that give a thorough background such as [41, 36]. I will not give an in-depth review of these basics. However, LTE is also quickly evolving, with LTE-Advanced (LTE-A) already being deployed commercially. The next generation of cell phone technology, 5G, is already being widely discussed. Even though Chapter 2 focuses on LTE, because LTE-A and 5G are new I present a review of these in Section 2.1. This section explores the impact these technologies are projected to have and proposes that a whole systems perspective with either a variety of TCP layer solutions or an intelligent TCP layer solution is necessary.

The second major issue covered is optimizing the video source. This section, covered in Chapter 3 deals with application layer protocols to reduce the bandwidth consumed by videos and the latency in video streaming. This Chapter looks at two broad types of solutions for this issue. The first solution is more efficient processing

through spatial tiling and selective streaming of the tiles. In this way, the bandwidth needed by the application can be greatly reduced. The second piece of the solution is moving the source from a distant cloud to the very edge of the network - the wireless AP. Both solutions are promising and can be implemented independently or in conjunction with each other.

2.1 Background

It is widely acknowledged that the current LTE user experience is much worse than what is advertised. Users experience lower throughput and higher latencies than what should be physically possible. The root of the problem is a mismatch between the design of the TCP layer protocol and the actual physical and MAC layer capabilities. While the congestion control algorithms for TCP have continued to evolve and improve over the past 30 years, one fundamental aspect has remained constant. These algorithms were all designed for relatively stable wired networks. A large body of recent works [48, 14, 28, 45, 44, 18, 30] show that LTE network conditions fluctuate extremely rapidly. My own experiments confirm extremely rapid fluctuations in both delay and throughput, as show in Figure 2.1. In this

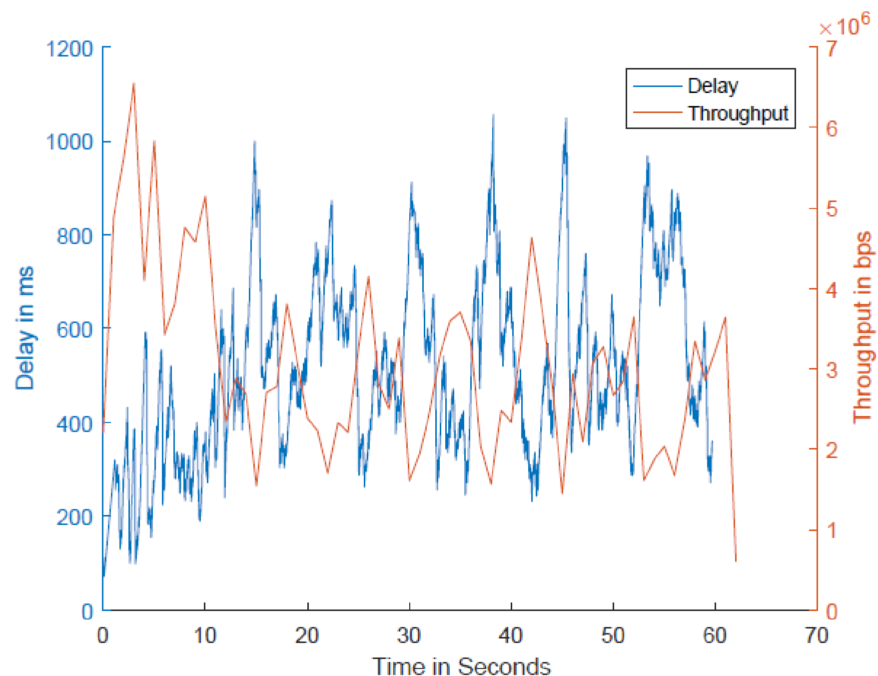

Figure 2.1: Throughput and Delay for LTE Network.

experiment, I tethered my laptop to a HTC One phone on Sprint's 4G LTE network. I modified the TCP Cubic kernel code to print the current delay, or round trip time (rtt), as experienced by the kernel. I ran Wireshark on my Laptop to capture the actual throughput at the receiver. I then ran an IPERF3 session between the server and laptop to attempt to saturate the link and determine the maximum throughput. Unlike wired networks (or even wireless protocols such as Wi-Fi), LTE physical layer characteristics change very rapidly, for several reasons including: mobility, additional users leaving or joining the network, weather conditions, and base station scheduling algorithms.

Because of these rapidly changing network conditions, and in an attempt to maximize cell utilization, cell phone carriers use very large buffers in their base stations. Both [18] and [47] attempt to measure these buffers. While each carrier has their own implementation, [47] shows that buffer size ranges from 400 to 4000 packets. Both [47] and my own tests show the delay from these large buffers can exceed 2 seconds. In addition, LTE employs very robust MAC layer protocols that prevent the TCP layer from experiencing any packet loss [18]. LTE Base Stations also maintain a resource scheduler that assigns available resources to different users. There are many potential algorithms to control this scheduler with different end goals in mind as show in [39]. However, one key component of all the algorithms is the traffic load. That is, if more traffic is presented to the base station scheduler more resources are assigned.

These factors are all far outside the normal behavior of wired networks that TCP was originally designed for. Loss based congestion control algorithms, such as Cubic, will never experience a loss over the LTE portion of the network. Therefore they continue to increase the sending rate to some maximum limit and incur very large delays. But if the delay grows long enough it can trigger the retransmit timer in Cubic which causes Cubic to begin fast retransmit of data that is not actually lost. This can lead to a sudden drop in goodput, as show in [30]. Loss based algorithms tend to have better throughput than other methods. Delay based congestion control algorithms, such as Vegas, seek to maintain the minimum delay. These algorithms are very good at reducing the delay, but frequently fail to utilize the available

bandwidth. Because LTE base stations schedule resources based at least in part on demand, they will not receive as many resources as a flow with a built up queue.

In addition to these two traditional TCP congestion control algorithms several new algorithms have been suggested, including: Verus [48], Sprout [44], CQIC [28], ACP [19], and PropRate [22, 21]. These algorithms all attempt to solve the problems presented earlier. However, all of these algorithms have at least some issues. Many were never tested over a live network, meaning the full interaction of their algorithm with the base station in the presence of other users was never tested. Many were developed on top of UDP instead of TCP. Base stations may treat UDP and TCP traffic fundamentally different, as shown in [39]. Many have good performance when it is the only congestion control algorithm used, but suffer poor performance when they compete with TCP Cubic. Most would not be easily implemented, as they would require both servers and end user devices to adopt an entirely new congestion control algorithm and may not be effective over wired networks. This would create a need for multiple congestion control protocols based on network conditions. Not only would that add a layer of complexity, but frequent switching between congestion control protocols takes time. When using HTTP the results of attempting to switch TCP congestion control algorithms are unpredictable because HTTP maintains a persistent open socket that often prevents TCP from switching congestion control algorithms.

Another possible solution is by implementing TCP middleboxes at the LTE base station (eNodeB). However, as [25] discusses in great detail, this is not trivial. First, this would require all service providers to uniformly alter their physical infrastructure and network topology. Beyond the difficulty of that, there are significant practical issues that would have to be overcome, most notably unexpected interactions with other middleboxes [25]. Finally, this solution would add significant security concerns as service providers would have to closely inspect every packet that traversed their infrastructure.

Given this background, I built a delay based congestion control algorithm as a modification to the current TCP Cubic Linux kernel implementation. This presents the easiest implementation to fix this issue while also allowing the continued

use of TCP Cubic for non-cellular based flows to the server. I used Matlab code based simulations to aid in the design. I utilized Amazon Web Server to build a custom Ubuntu Linux kernel with my modifications. I ran all experiments over a live cellular network, primarily utilizing Sprint's 4G/LTE network near Madison, WI. To better understand the problem and the performance of my algorithm, I conducted extensive comparisons between my algorithm, TCP Cubic, TCP Vegas, and Verus.

LTE-Advanced and 5G

While the rest of this chapter focuses on improving TCP throughput over existing 4G LTE networks, it is important to mention that LTE Advanced is already being deployed across the US and around the world. However, to my knowledge there has not been any widespread, systematic study of how the new features introduced in LTE Advance will affect the TCP layer throughput. Some of the key components of LTE-Advance include: carrier aggregation, advanced MIMO, HetNets, enhanced receivers, LTE in unlicensed spectrum and new types of LTE transmissions. Proponents of these features claim they will all work hand in hand to greatly boost throughput and reduce latency. I will briefly review these features and highlight some existing works demonstrating their effectiveness in an attempt to give a qualitative understanding of how they will affect user perceived (TCP layer) throughput and delay.

Carrier Aggregation

LTE-A allows for up to 5 individual carriers to be aggregated together. Each individual carrier may be up to 20 MHz, allowing for a total aggregation of up to 100 MHz. LTE-A provides for both intra and inter band aggregation with either contiguous or non-contiguous bands. This is especially important considering how much of the bandwidth in the US has been acquired by providers in a very piecemeal way. Carrier aggregation allows for higher peak data rates and lower

latency. Aggregation of two carrier frequencies doubles the burst rate for all users in the cell, which can reduce over-the-air latency by about 50% [32].

Advanced MIMO

Another improvement in LTE-A is the implementation of more advanced MIMO systems. LTE-A provides for up to 8x8 multi-user MIMO on downlink communications. It provides for up to 4x4 single-user MIMO on the uplink. One of the most immediate improvements is the increased antenna diversity on the UE. 2x2 MIMO is used in most current mainstream commercial systems. Adding two more antennas only to the UE, making a 2x4 MIMO on the downlink, can give up to 1.7x receive diversity gain [32]. This is significant because it does not require any change to the service provider's infrastructure and can be implemented immediately on new UE devices.

Coordinated Multipoint (CoMP)

Coordinated Multipoint, or CoMP, allows the optimization of transmission and reception from multiple distribution points in a coordinate way. It can work in both homogeneous networks or heterogeneous networks. Most uses in homogeneous networks are envisioned to provide improvements at the cell edge, especially where two cells coverage may overlap. The heterogeneous use will be explored more in the next section. CoMP allows for both joint transmission and/or reception among the base stations [35].

HetNets

The idea of Heterogeneous Networks relies on several of the previous technologies. The main concept is to add smaller (micro, pico, femto) cells to improve peak performance in particular areas while still maintaining the traditional wide area macro cell coverage. A macro cell, plus 4 Pico cells can offer up to a 2.8x improvement over a macro cell only [32]. In addition to relying heavily on advanced MIMO and

CoMP, HetNets also require enhanced device receivers and will most likely utilize unlicensed spectrum, which will be covered in the next two paragraphs.

Enhanced Receivers

One of the key challenges of HetNets is the additional interference caused by multiple cells. LTE-A tackles that problem primarily by focusing on enhanced interference cancellation at the receiver (UE). 3GPP release 10 and 11 provided standards for interference cancellation of synchronizing reference signals, common reference signal, and the primary broadcast channel. Release 12 also includes standards for interference cancellation of the data channel. In a live demonstration conducted in 2014, Qualcomm showed the enhanced receivers can double the throughput for pico cells [32], though larger cells had less benefit.

Unlicensed Spectrum

Carriers may aggregate both licensed and unlicensed spectrum, primarily in very small cells, to provide boosts in performance. There are also provisions to allow for Authorized Shared Access or Licensed Shared Access, primarily on government owned, sparsely utilized frequency bands. One important thing to note is that while these schemes bring new bandwidth to LTE, these frequency bands already are in use by other devices, most notably Wi-Fi. This creates additional contention for limited bandwidth.

New Transmission Types

LTE-A introduces several new transmission types. The first is LTE-Direct which provides for device to device proximity awareness. LTE-Direct is a device based, connectionless discovery protocol with a range up to 500m. LTE-Direct is projected to have a negligible capacity impact [32]. LTE-Machine Type Communication (MTC) is a set of new physical and MAC layer protocols to allow for a massive amount of IOT devices to communicate without completely congesting the radio network [35]. The new transmission type with the largest impact on the TCP layer is LTE-Broadcast.

LTE-Broadcast, much like its name implies, is a broadcast service carrying a single content stream to multiple users. Qualcomm conducted a study in 2014 which demonstrated that LTE-Broadcast offers significant gains when multiple users are receiving the same content [32] . More surprisingly, it also shows improvement to network capacity for even one user. This is because LTE-Broadcast removes much of the signaling overhead associated with traditional LTE communication.

LTE-A Implications to TCP Layer

Although US cell providers are quick to boast of their growing deployment of LTE-A across the US, I have been unable to find any comprehensive studies on the actual performance improvements. One reason may be that LTE is a continuous evolution of technologies with a very piecemeal introduction of new features. Some studies have been done on individual components of LTE, though not in a live commercial environment. DOCOMO, the major mobile phone operator in Japan, conducted several detailed experiments focused on the effects of CoMP, advanced MIMO and Carrier Aggregation in 2012 [20]. They were able to achieve a throughput of about 600 Mbps throughput while traveling at 10 km/h using carrier aggregation. However, there were still large variations in throughput and the throughput depended heavily on proximity to the base station. One major shortcoming of DOCMO's study was that the effect of multiple users was not well explored. They did one indoor test with two users to explore a 4x2 MU-MIMO systems throughput. The total throughput for all users can exceed 800 Mbps but they also found that some users can experience lower throughput than others and the throughput can still vary by over 100 Mbps in about 1 second.

Overall, LTE-A should provide both an increase and a more consistent physical layer throughput than current LTE. This should improve the TCP layer performance in general. However, even with these improvements, LTE-A will still not equal the stability or capacity of a wired network. Furthermore, LTE-A will be deployed incrementally in terms of both location and features supported. Because of these factors, LTE-Advanced, much like LTE, will greatly benefit from either a range of

TCP layer solutions, or an intelligent TCP layer solution, to optimize the user experience (maximize throughput and minimize delay) in a wide range of environments and usage scenarios.

5G Expectations

There are many new aspects to 5G. Indeed, all sources stress that 5G will be an entirely new type of network and not an evolution of existing networks. This paper will not attempt to give even an overview of all the possible 5G technologies. Eventually, when all the currently proposed 5G technologies are implemented, they may be able address the majority of the 5G requirements. However, the extent that the requirements are met, or even if they are all met, is still a very unanswered question.

As the NGMN Alliance reported in 2015 [29]:

> To illustrate, the network capacity of a radio network depends on the spectral efficiency, spectrum bandwidth and cell density. Shannon's capacity equation is a fundamental constraint on the spectral efficiency performance and implies logarithmic dependence of channel capacity on the signal-to- interference and noise ratio (SINR). Interference mitigation and coordination techniques could improve the effective SINR, especially at low SINR regions, thereby improving the system spectral efficiency. Massive MIMO could also improve the SINR through narrow beam-forming, pushing the system closer to a noise limited environment. If the system could realize high SINR regions in a wider area through use of such technologies, advanced coding and modulation schemes may provide more effective gains. Furthermore, flexible and full duplex, as well as schemes which reduce the amount of guard bands and overhead, may improve the overall spectral efficiency. However, given the theoretical limit and technologies, it seems unrealistic to assume order of magnitude improvements.

Whether or not the implementation of 5G will achieve all the current goals, it is worth a quick overview of those goals. 5G looks to be a very heterogeneous environment with multiple access technologies and multiple performance needs [40]. It seems likely that 5G will have a wide range of flexible and dynamic UE's with a range of quality of service (QoS) requirements and node capacities. Much like LTE-A, 5G will support multiple bands with aggregation of flows. All these varieties point to the need for either multiple TCP layer protocols, or an intelligent, Physical Layer informed TCP layer protocol. This raises two major questions: What is the feasibility of accessing physical layer information in 5G systems? And is the trade off between increased complexity and increase performance worth it?

The main challenge to accessing the physical layer information in 5G is the very heterogeneous environment with multiple access technologies. However, this is already an issue for other reasons and so current proposals call for 5G to be a much more intelligent network. In addition, MIMO and CoMP transmissions will rely on the availability of channel state information to realize their full potential [29]. These factors natively support exposing UE measurements to higher layer protocols.

To gain a better understanding of the potential trade off between increased complexity between the network layers and increase performance, I conducted a series of experiments focused on one key aspect of 5G, extremely low latency. To this end, I modified the existing TCP cubic congestion control algorithm in the Linux Kernel by adding a simple delay based TCP reduction algorithm similar to that described in Section 2.3. Specifically I used Equation 2.2 to calculate the extra packets added to a queue. I then directly subtracted the value of diff from the current sending window. I made no other changes to the standard TCP Cubic congestion control algorithm. I tested this modified TCP congestion control algorithm on a live Wi-Fi network over a variety of mobile paths. 2.2 shows a plot of the round trip times (rtt) with TCP Cubic on the left and my modified TCP Cubic on the right. Over the course of several experiments I was able to maintain the same average throughput with a 20-35% decrease in rtt and a 60-70% decrease in jitter. This demonstrates that significant performance gains in one aspect (latency) can be achieved with very minor increases in complexity. 5G would benefit from either a

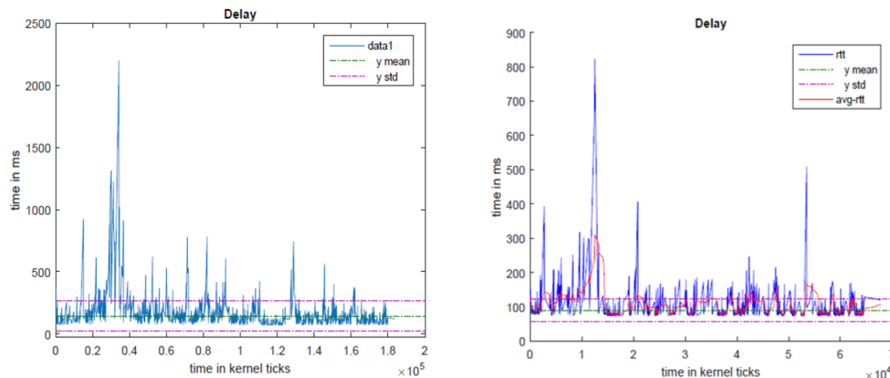

Figure 2.2: Standard TCP Cubic (L), Modified TCP Cubic (R).

variety of TCP layer congestion control algorithms or a more intelligent, physical layer informed TCP congestion control algorithm.

2.2 Relationship Between Throughput and Delay

It is clear from recent works such as [18, 48, 47] that cell phone providers maintain long individual queues at the eNodeB. What is a little less clear is the impact of the size of these queues, especially in competition with other flows. If the queue gets too large, the flow will suffer from buffer bloat and unresponsiveness to changing channel conditions. On the other hand, if the queue is empty, the base station will not assign any more physical resources to that flow, leading to underutilization of the available resources and reduced throughput.

While each flow may have its own queue in the eNodeB, they must share the same physical radio resources for transmission. Zaki shows in [48] that for 3G networks, when operating near saturation, competing flows do impact the round trip time. I attempted to confirm this for LTE networks, with mixed results. Table 2.1 shows the results of a simple experiment to test this claim for LTE. I utilized two separate phones, both HTC One's utilizing Sprint's LTE network. One phone connected to a public IPERF3 server and was either on for the whole time or off for the whole time. The other connected to my Amazon EC2 web server running as an

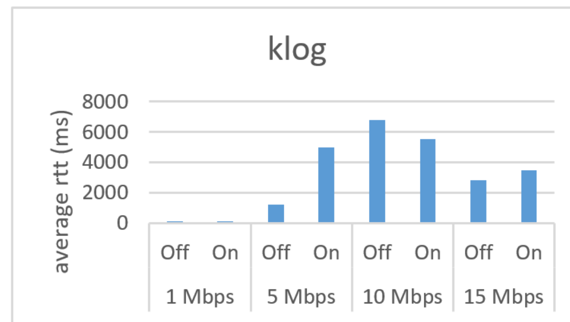

Table 2.1: Impact of Competing Flows on rtt.

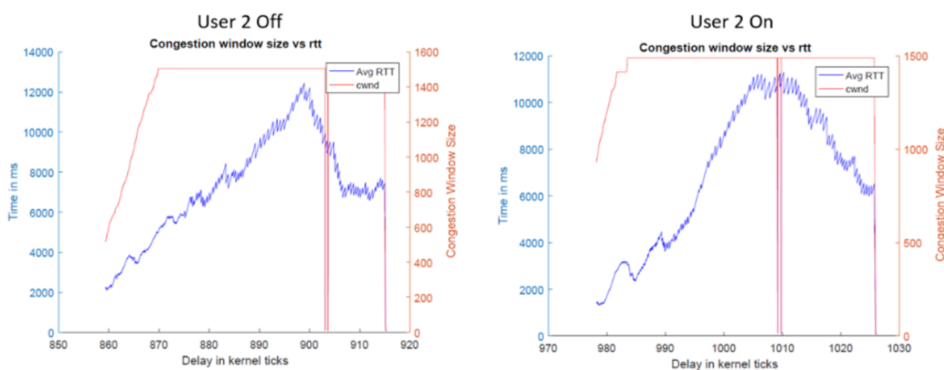

Figure 2.3: Impact of Competing Flows on Congestion Window Size.

IPERF3 server to record the rtt reported to the kernel. Both IPERF sessions attempt to send at the same rate. It is clear from the 1 Mbps results that when the channel is very underutilized a second user has no impact on the first. It is also clear that when the channel is heavily utilized, the delay increases greatly. But it does not seem there is any direct correlation between the second user being on or off and the average delay. Figure 2.3 shows the actual rtt and sending window size from the kernel during the 10 Mbps test. What is clear is that regardless of other users queue length, the sending window increases rapidly to a very high value and then stays there. The delay follows a similar pattern where it generally increases with the increasing congestion window. In these cases, the delay can increase to over 10s before it appears there is a loss event and the delay begins to reduce.

To get a better understanding of the relationship between the channel capacity

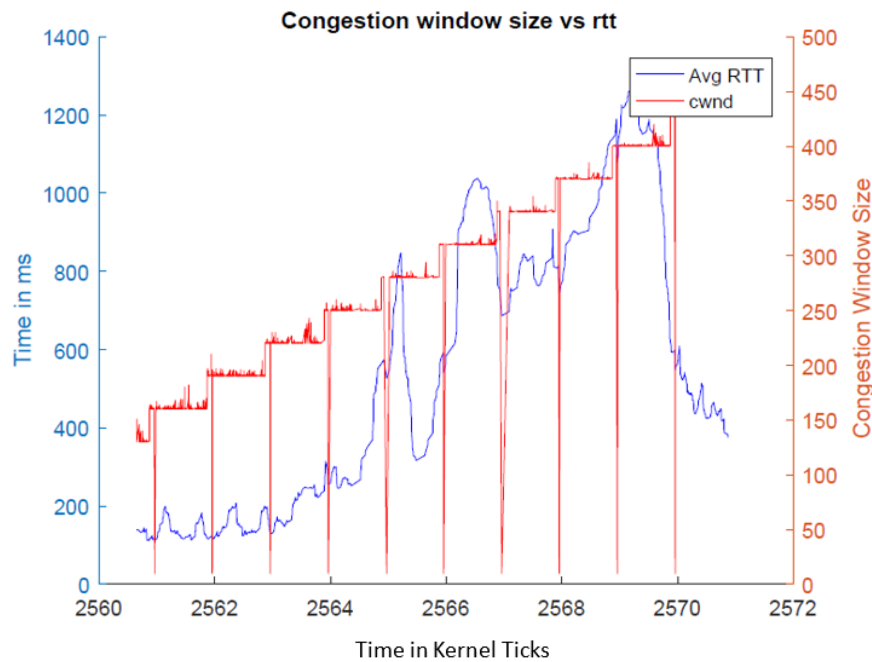

Figure 2.4: Congestion Window Size Impact on rtt.

and the delay, I designed another experiment. In this experiment I directly controlled the sending window size on the server. I increased the window sending size every second by a fixed amount. I then recorded the sending window size and delay. I did this many times, varying both the starting size of the congestion control window and the magnitude of the increments. 2.4 is one of the results which demonstrates the general trend I observed throughout the results. Increasing the size of the congestion window below a certain threshold has no impact on the rtt at all. As long as the rtt remains relatively stable, the actual throughput is equal to the sending window divided by the rtt. This means that, in Figure 2.4, the channel capacity was initially underutilized. During the first three increments the channel was underutilized so the eNodeB scheduler was able to increase the throughput to the end user device (UE) and delay remained constant. At about time 2564 the delay begins to rapidly increase, and continues to increase with the increasing congestion window size. This indicates that the channel was fully utilized, and the increase in

sending rate only filled the buffer at the base station.

These experiments show that LTE base stations do maintain large individual queues. The length of competing queues does not directly impact the throughput of a given flow. However, there are limited physical radio resources available, and once the channel is congested the queue will only increase with increasing sending window.

2.3 Design of Congestion Control Algorithm

The previous observations lead to the following design guidelines. Principle A is, that we should send at a rate great enough to maintain a queue at the base station. This ensures we always have traffic waiting to be sent if more physical resources become available. Principle B is, we should send at a rate slow enough that we do not build a large queue and suffer from buffer bloat. This will ensure we slow down if other users join the network and less physical resources are available. TCP Cubic is good at Principle A, but not at Principle B. TCP Vegas and Verus are good at Principle B, but not good at Principle A since both intentionally try and maintain the minimum rtt which means there is no queue at the base station.

I created a delay based congestion control algorithm that aims at maintaining a fixed length rtt greater than the minimum observed rtt. It is important that the delay be significantly greater than the minimum delay to ensure the base station experiences enough demand for network resources that it will schedule more resources if they are available. To determine how long of a queue we should maintain I first did a large group of experiments to determine a normal rtt for an un-congested link. I found that a rtt slightly under 90ms was fairly typical using a ping test over Sprint's LTE network. I also wanted to ensure that the maximum delay increase or decrease was not greater than the average human response time to visual stimulus. This would ensure that the worst case scenario for changing network conditions would not be noticeable by someone using live video streaming. Numbers for human response time vary greatly, but a reasonable number seems to be about 270 ms. For these reasons, I focused on fixing the delay to 3 or 4 times

the minimum delay. That would give a maximum delay of either 270 ms or 360 ms, respectively. However, we are actually concerned with the change in delay (delta), not the absolute delay. The delta is 180ms and 270ms respectively. With these factors in mind, I tested several possible factors from 2 to 10 over a simulated LTE network. I utilized Linux NETEM commands to emulate the throughput and delay recorded during live experiments. This created a basic simulation using wired and Wi-Fi networks to conduct tests on some of the fundamental aspects of the algorithm during the design stage. These simulations showed that using a factor greater than 4 did not lead to any significant increase in throughput, but did lead to higher delay. I then conducted several tests over live LTE networks and settled on using the factor of 4 as it allows the base station to experience the longest queues while not increasing the rtt enough for a typical user to notice.

TCP Cubic already has built in mechanism to keep track of the minimum delay in the function bictcp_acked. I created a variable called delay_threshold that simply multiplies the minimum delay experienced by TCP by a factor of 4. I created two additional variables; one to keep track of the physical resources available (prb_avail) and the other to calculate the number of packets we have added to the queue above the threshold (diff). I created three states based on the current delay within the function bictcp_acked.

The first state is when we exceed the delay_threshold, that is:

$$\mathtt{delay} > \mathtt{delay_threshold} \tag{2.1}$$

In this state we are sending too fast and should slow down. To calculate how much we should slow down by, I used an approach similar to that of TCP Vegas. We need to calculate how many additional packets have been added to the base stations queue when it exceeds the threshold delay. The current sending rate is found by multiplying the sending window by the delay. The ideal sending rate is found in a similar manner except using the delay threshold instead of the current delay. We then take the difference of the two, and divide by the delay threshold to find the

number of packets we have added to the queue. The exact equation is:

$$diff = snd_cwnd * (delay - delay_threshold)/delay_threshold \quad (2.2)$$

After the algorithm calculates diff it also sets the prb_avail to 0.

The second state is when we have not exceeded the delay threshold, but the current delay is greater than 80% of the delay threshold. This state was not in the original algorithm, but I found that without it the algorithm frequently overshot the delay threshold and was not very stable. This is because delay, while a reliable indicator of network conditions, is delayed feedback. In other words, it takes time (at least one rtt) for a change in the sending rate to effect the measured delay. Because of this, I had to build some hysteresis into the protocol. In this state, diff is set to 0. The prb_avail is calculated by finding the difference in delay_threshold and current delay, divided by the delay threshold. The result should then be multiplied by 100 to give the prb_avail as a percentage. Instead, the algorithm scales down the prb_avail and multiplies by 40. This factor ensures the congestion control window growth slows down quickly enough to avoid frequent overshooting. It was chosen because it is half the factor of the third state, as described in the next paragraph. The exact equation is

$$prb_avail = 40 * (delay_threshold - delay)/delay_threshold \quad (2.3)$$

The third state is when the delay is less than 80% of the target delay. During this state, the algorithm calculates the distance from the delay threshold and sets that as the prb_avail in a similar to the second state. The exact calculation is:

$$prb_avail = 80 * (delay_threshold - delay)/delay_threshold \quad (2.4)$$

The multiple factor is 80, rather than 100, to slow down the probing slightly. After I realized that using the full 100% of the resources was unstable, I repeatedly tested the algorithm with slightly lower values. I found that at around 80% the algorithm still grew rapidly, but was much more responsive to the changes in delay. Diff is

again set to 0.

While the delay, delay threshold, prb_avail and diff are calculated in the function bictcp_acked, the actual probing up or decreasing takes place in the Cubic function bictcp_cong_avoid. If the prb_avail variable is greater than zero (we have not overshot the target delay) the probing function calculates the number of acks needed (ca->cnt) to increase the sending window using the prb_avail. This variable, ca->cnt, is the same one changed by Cubic to control the growth of its sending window. The smaller it is, the faster the sending window will grow, and vice versa. Our growth algorithm is controlled by the equation:

$$cnt = 1/prb_avail^2 \tag{2.5}$$

This equation grows the sending window very rapidly when we are far away from the target delay, but also slows down rapidly as we approach the target delay. We also ensure that the minimum value of the ca->cnt is 2. This is also the minimum value of Cubic, and ensures we do not increase the sending window more than 50% in one rtt. This is important because it takes at least one rtt for a change in the sending window to effect the delay.

When the prb_avail is zero (the delay has increased beyond the delay_threshold), we use the diff value and directly change the sending window by subtracting diff. This results in an immediate decrease in the sending window. We also set ca->cnt to 100 times the current send window. This ensures that the window does not immediately grow again after we experience excessive delay. The algorithm may still receive many acks while in this stage and it is important that we do not rapidly grow the sending window.

A little more functionality needed to be added to the cubic function bictcp_acked to deal with some unexpected consequences of directly modifying the sending window size. During initial testing I observed that the sending window would decrease by diff, but then continue to rapidly decrease all the way to the minimum window size. The problem is that as soon as the window size decreases, there are more packets in flight than there should be. This triggers a packet loss event in

Cubic, and therefore resets the window size to the minimum. Therefore, I added a variable to keep track of the sending window when it is changed by our algorithm. I then force the sending window to stay the size that our algorithm calculates and not reduce further.

In early trials, this performed quite well; the throughput was very comparable to Cubic and the delay was greatly reduced. However, I observed another problem. Once the queue has jumped above the delay threshold, it takes a noticeable amount of time for it to drain even after we decrease the sending window. In other words, the rtt acts like a delayed feedback but my algorithm continued to decrease the send window very rapidly. Therefore, I added a simple counter to the reduction scheme that ensures we wait some amount of time to observe the effects before reducing the sending window again. The counter is incremented every time the function bictcp_acked is called. If it reaches the count value (currently set to 100) it will reset the count and allow another decrease. On the other hand, if the delay drops below the delay threshold, the algorithm returns to the probing up state and resets the counter. This feature ensures we only reduce the sending window by the amount we added to the queue and do not drop the sending window to far. This feature was even more important in 3G performance. During tests over 3G, I noticed a very periodic momentary spike in the rtt (about every 5s). This only occurred over 3G, there was no comparable periodic spikes in LTE. These spikes in delay over 3G were compounded by multiple decreases in the sending window within a few kernel ticks. The top figure in Figure 2.5 shows the effects without the count feature to delay sequential reductions. While the sending window is quick to grow again, it is clearly not optimal. The bottom figure in Figure 2.5 shows the results of adding this feature. There are still somewhat regular spikes in the delay, but the algorithm adjusts only once and maintains a much steadier sending window and delay and therefore a much more stable throughput as well.

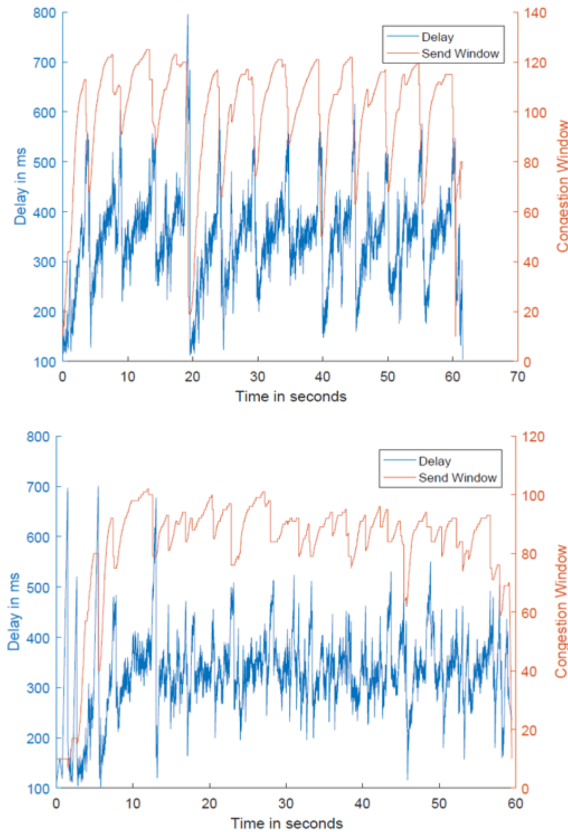

Figure 2.5: Congestion Window Without and With Delay Count.

2.4 Experimental Results

To verify that my congestion control algorithm does offer an improvement in performance, I compared it to TCP Cubic, TCP Vegas, and Verus. For each of the four congestion control algorithms I was comparing, I performed two sets of tests. The first set was the single user test. In this test, I used the Amazon Web Service server running Ubuntu Linux. I modified the TCP Congestion Control Kernel to print the rtt and current send window. I also added my congestion control algorithm to the TCP Cubic code with a sysctl variable to turn the modification on or off. I tethered my laptop to the phone and utilized the phone's LTE connection as the only Internet connection. In this way, I was able to use Wireshark on my laptop

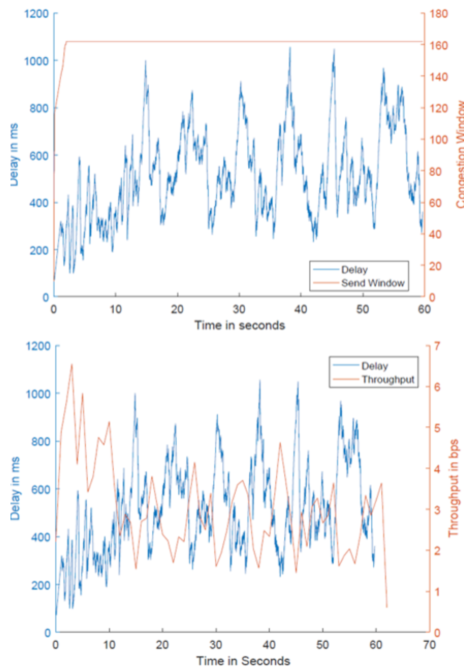

Figure 2.6: TCP Cubic: Congestion Window and Throughput vs Delay.

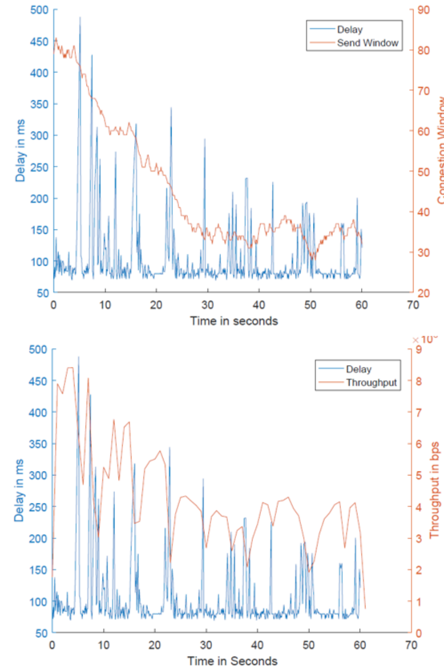

Figure 2.7: TCP Vegas: Congestion Window and Throughput vs Delay.

to capture the actual throughput on the receiver side. I then used IPERF3 to send traffic from the server to my laptop. Each test lasted one minute. Figure 2.6 shows one of the results using the Cubic algorithm. It is obvious that Cubic not respond to changing channel conditions and as a result has very high delay that generally increases. Figure 2.7 shows one of the results using the Vegas algorithm. While TCP Vegas does a good job of keeping the delay to a minimum, It suffers from a continually decreasing throughput. It also does not place enough demand on the base station in order to be assigned additional radio resources. Figure 2.8 shows one of the results of the Verus congestion control algorithm. Verus currently runs as a standalone C program over UDP. The source code comes with a built in network test similar to IPERF and a built in plotting function. The resulting plot must have been heavily smoothed, as all the other rtts were never that constant. Still, you can see from the graph that, much like Vegas, Verus does a god job of minimizing the

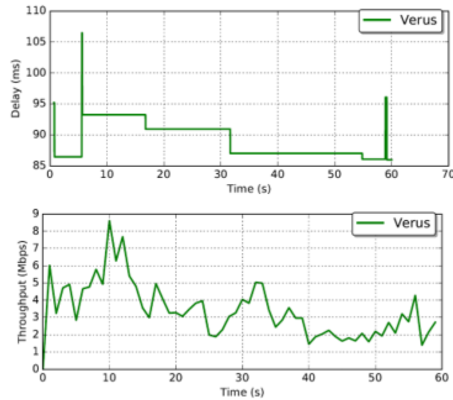

Figure 2.8: Verus: Throughput and Delay.

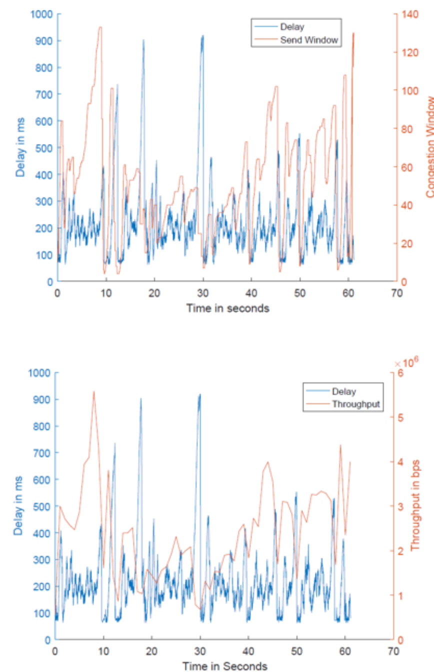

Figure 2.9: My Algorithm: Congestion Window and Throughput vs Delay.

delay but also has a continuously decreasing throughput. Figure 2.9 shows one of the results using my delay based congestion control algorithm. It is immediately obvious from the figure that it behaves differently from all the other congestion control algorithms. The algorithm grows the sending window rapidly when the delay is less than the delay threshold, but also is very fast to react by reducing the sending window when the delay increases beyond the delay threshold. In this way, it combines the high throughput of Cubic with the low delay of Vegas. While my algorithm and Verus both outperform TCP Cubic and Vegas, my algorithm generally outperforms Verus in throughput, though by design it has higher delays.

The second test I performed was utilizing two users. The first user was set up exactly the same as in the previous test. The second user consisted of an HTC One phone in the same location connecting to a public IPERF3 server utilizing the standard TCP Cubic congestion control algorithm. The second user was on from

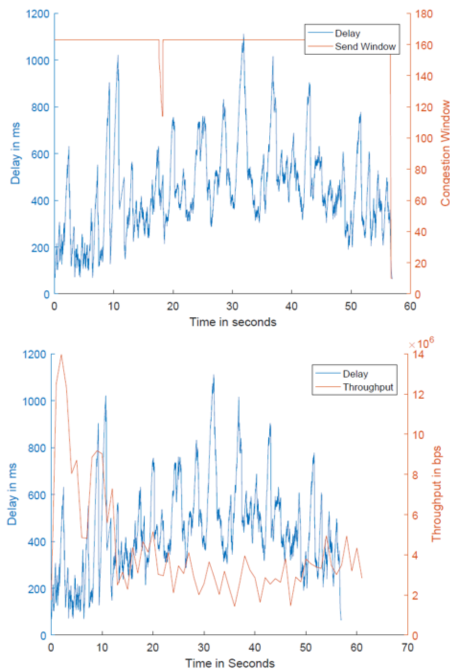

Figure 2.10: TCP Cubic with Two Users: Congestion Window and Throughput vs Delay.

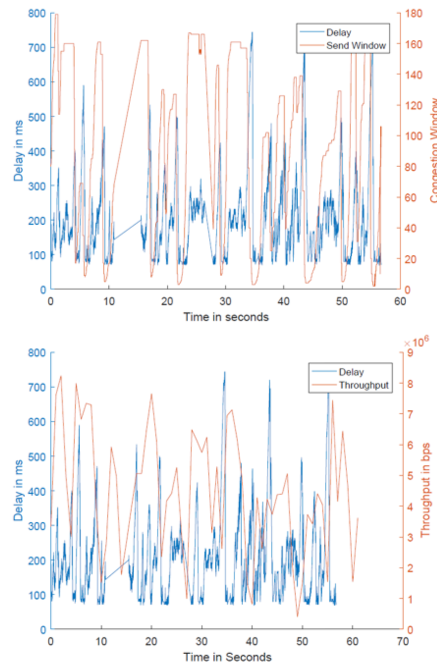

Figure 2.11: My Algorithm with Two Users: Congestion Window and Throughput vs Delay.

about time 20 to time 40. This provides a good snapshot of how the congestion control algorithm will perform under two extremes: a sudden decrease in network resources and a sudden increase in network resources. In general, I did not observe a drastic change in performance for any of the algorithms. This was somewhat surprising, especially because prior tests on 3G and simulations both displayed a much more noticeable effect. The primary cause of this is probably the base stations radio resource scheduler. While base stations scheduling policies are propriety, most use a proportional fair scheduler based algorithm [1, 46, 45]. Because of this, many of the fairness issues at the TCP layer experienced over traditional wireline networks are removed at the physical or MAC layer. However, I still observed some general trends. Figure 2.10 is one of the results from the two user test utilizing TCP Cubic and demonstrates the first general trend; that is, bandwidth is initially high but then decreases and remains low. In fact, this happened about in about 77% of

the experiments lasting 20 seconds or more regardless of what congestion control implementation was used. This indicates that the base station scheduler may take flow length into account when scheduling and gives preference to new flows. Figure 2.11 is one of the two user results for my delay based congestion control algorithm. It demonstrates that my algorithm is capable of increasing the bandwidth at any point in time. This is very different from the the other algorithms, which all consistently demonstrated little to no increase in bandwidth over time. Figure 2.12 is one of the results from the two user test utilizing TCP Vegas as the congestion control algorithm. TCP Vegas was affected more than the other algorithms by the second user. Because a second user adds contention for limited resources, there tends to be an increase in delay for the first user. TCP Vegas therefore reduces the size of the sending window even further. Even after the second user turns off, it does not have a good mechanism to probe for newly available bandwidth. TCP Vegas is not widely used in traditional networks because it cannot compete with other TCP congestion control algorithms as they probe for more bandwidth. In LTE, some of the problem is removed because the proportional fair scheduler at the base station ensures that each user gets a fair amount of resources. However, TCP Vegas still suffers because it does not have a good mechanism to probe upwards even after the contention is removed. Figure 2.13 is one of the results of using the Verus algorithm. Again, it suffers a similar problem as TCP Vegas. In an effort to achieve the minimal delay possible, the algorithm is very slow to probe upwards.

2.5 Further Analysis

Several trends were apparent throughout all of my testing. First, TCP Cubic is more aggressive at utilizing available bandwidth, but suffers from long delays and is not very responsive to changes in network conditions. TCP Vegas and Verus are both less aggressive at utilizing bandwidth, but have much lower delays. My delay based congestion control algorithm is more aggressive at utilizing network bandwidth than any of the other congestion control algorithms, while not suffering the buffer bloat issues that TCP Cubic has.

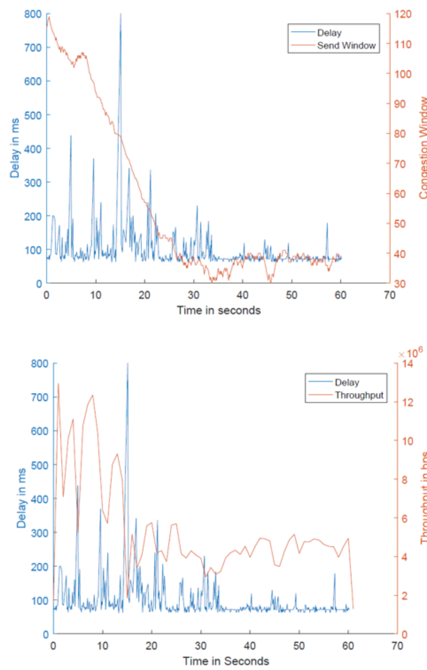

Figure 2.12: TCP Vegas with Two Users: Congestion Window and Throughput vs Delay.

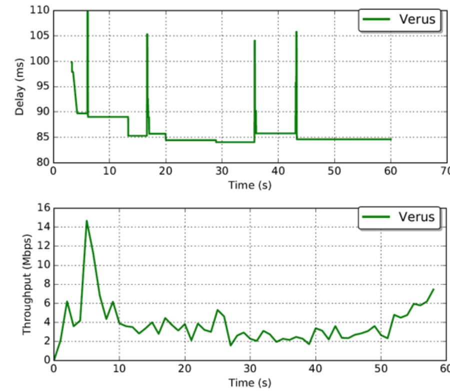

Figure 2.13: Verus with Two Users: Congestion Window and Delay.

Second, I observed that the network conditions are rarely stable for more than 5 - 10 seconds. This fits with the existing body of research that shows that LTE network conditions fluctuate rapidly. However, my experiments show that the initial 10 - 15 seconds have a much higher throughput that then generally decreases. This is most likely due to the specific proportional fair scheduler utilized at the eNodeB. It could also be caused by the interaction of the base station scheduler and TCP congestion control. When the connection first starts there is very little delay so the sending window increases rapidly. Because of that, the buffer at the base station begins to fill and then the delay increases rapidly. This causes delay based algorithms to reduce the sending window and therefore reduce the overall throughput. It causes TCP Cubic to develop buffer bloat so that even though the sending window remains large, the time it takes to receive an ack and send the next packet reduces the actual throughput. Allowing the delay to remain larger than

the minimum, but not so large as to suffer buffer bloat, offers the best protection against this phenomenon.

Third, I observed that in general LTE users with different congestion control algorithms do not have the same fairness problems that occur in traditional network environments. In shorter tests, lasting 20 seconds, there was a more noticeable difference in performance when a TCP Cubic user joined half way through than in longer tests that lasted 60 seconds. This may simply be the result of the second result described above. Regardless of the cause, delay based congestion control algorithms perform worse in competition with TCP Cubic. While the effects may not be as severe as in traditional networks during any duration test, these protocols still suffer degraded performance when competing with TCP Cubic. TCP Vegas suffers the worst of these consequences, while my delay based congestion control algorithm experiences the least negative consequences.

Fourth, I confirmed that Verus does generally outperform both TCP Cubic and TCP Vegas. But my delay based congestion control algorithm generally outperforms all three. However, the performance gain is not the only advantage. I found that switching between TCP Cubic and TCP Vegas in the Linux Kernel does not happen very rapidly. For some time after the switch new flows continue to use the old congestion control algorithm. This is especially true for HTTP traffic, as noted earlier. I found that the only reliable method to ensure the new congestion algorithm is used is completely shutting down and restarting the interface in question, but that is not feasible in production networks. Even closing the TCP socket and opening a new one in the Kernel did not guarantee the new socket would utilize the new congestion control algorithm. The exact amount of time it took to switch depended on higher layer protocols, but was the worst for HTTP which accounts for the majority of Internet traffic. In other words, it would not be feasible for a server to run TCP Cubic for traditional network topologies and switch to TCP Vegas for LTE. Verus, on the other hand, is not even built on TCP nor is it a true kernel implementation. While it may be possible to use it in a production environment, it would require additional applications to run on the server and be able to intercept all network traffic and determine which are LTE based. This is a significant additional obstacle

to a real implementation. In fact, I did not find any previous works that are built and tested in the actual TCP stack of the Linux Kernel and tested over a live network.

Fifth, I confirmed that delay based congestion control algorithms generally outperform traditional window based algorithms that use loss as the congestion trigger in LTE networks. This has also been shown in previous works. However, these types of algorithms have not been adopted because in traditional network topologies they cannot achieve a fair rate when competing against other congestion control algorithms. While this impact is reduced in LTE (as discussed earlier) the real key to overcoming the fairness issue is changing the desired delay. All previous works have attempted to consistently achieve the minimum delay which always causes them to reduce their sending rate when competing with TCP Cubic. This also reduces the amount of traffic at the base station for the flow using a delay based congestion control algorithm so the base station will assign proportionally less resources than to a competing Cubic flow. This creates a type of vicious cycle where TCP Cubic will get more resources and other delay based congestion control algorithms less with each iteration. By setting a delay threshold that is higher than the minimum, we can fill enough of the buffer to cause the LTE base station to schedule more resources (and thus compete with TCP Cubic) while not suffering the negative consequences of buffer bloat.

2.6 Backward Compatibility

In our mobile world the amount of data transferred over cellular networks is very large, and only going to increase. LTE communication brings new challenges that traditional networks did not face, including rapidly changing channel conditions and unintended network shaping by the base station. Understanding these challenges and how different TCP congestion control algorithms interact with the LTE system is key to being able to maximize throughput for mobile users. The benefits of achieving higher throughput to users are obvious. Similarly, content providers are competing to provide the best experience to the user. Any potential gain must be both backwards compatible and simple enough to implement to justify any

performance gains. I have demonstrated that substantial performance gains can be achieved through a relatively simple modification of the existing TCP Cubic Kernel. This implementation required less than 50 lines of code added to the TCP Cubic Kernel but resulted in significant improvement over 3 existing congestion control algorithms. To fully implement my algorithm, the server side would need a simple kernel upgrade. The client side would also need a simple upgrade to add a flag in the TCP packet header for LTE use. That is the only change needed on the end user side. My experiments over a live network utilizing the actual TCP Cubic Linux Kernel demonstrates that it is feasible to implement a new congestion control algorithm that offers significant performance gains over previous widely implemented congestion control algorithms.

3 IMPROVING VIRTUAL REALITY VIDEO STREAMING

3.1 Background

Virtual Reality (VR) videos are on the cutting edge of media content. Virtual Reality is a completely immersive experience, where the user's entire visual field is limited to the display. Streaming VR videos requires extremely high bandwidths. Not only does the video have a very large field of view (a full 360° sphere), it must also be shot in high resolution and, for 3D, must have two complete videos from slightly different perspectives. This results in large video files, requiring bandwidths on the order of 60 Mbps or more to stream these videos. The high bandwidth demand for streaming VR videos is an even greater hurdle when multiple users are viewing the video over a shared contention medium like Wi-Fi. This fact has greatly hindered true VR streaming from achieving wide use so far. However, 360° video streaming, a sort of pseudo-VR experience, is growing at an incredible rate. In 360° video streaming, you still have access to a complete 360° sphere viewing area. However, it is not a 3D experience, and it is often recorded in much lower resolution. YouTube began streaming 360° videos a little over a year ago and there are already over 410,000 videos hosted there alone. Being able to reduce the bandwidth needed to a reasonable rate is key to the success of true VR video streaming. Several existing works have explored different methods of improving video streaming, including [6, 38, 27, 12, 13, 34, 31, 17, 4, 24, 2, 45]. While optimizing for video streaming has been studied at length, I examine two approaches that have not be implemented in any production systems to date. The first solution explores methods to limit the scope of the video that needs to be transmitted and to reduce the overall size of the video without reducing the users quality of experience. The second major section looks at the effect of moving the video storage from a distant cloud server to the extreme edge of the network. Both of these solutions have great potential for advancing VR video streaming performance over wireless links and they can be easily combined to further increase performance gains.

3.2 Improving VR Streaming Through Video Processing

One of the key challenges to streaming VR video is the large bandwidth required. VR videos comprise a full 360° degree field of view and must contain two videos to enable depth perception. It is widely accepted that 4K video resolution is the minimum acceptable resolution for head mounted displays used in VR applications [12]. These factors combined create the need for Wi-Fi streaming rates around 60 Mbps or higher. While this is well within the capacity of the latest 802.11ac access points for a single user, it is marginally greater than the average US home's stable end to end throughput. Speetest.net reports the average broadband download throughput in the US is about 55 Mbps. However, the bigger challenge is with mobile users. Speedtest.net reports the average US mobile download throughput is about 20 Mbps, well below the minimum requirements for VR videos. In this section, I will detail several existing works on how to tackle this problem, along with experimental results demonstrating both the benefits and challenges of this approach.

Dynamic Adaptive Streaming

One of the most common approaches for reducing the required bandwidth while streaming bandwidth-intensive multimedia applications is to utilize adaptive streaming [12]. Adaptive streaming is a process where the quality of the video being sent from a server to a client can be altered in real time. In this way, a client can maintain the highest possible video quality without exceeding the actual network throughput. In practice, there are some limitations to a dynamic adaptive streaming system. In most systems, the server has a small number of resolutions available to choose from. There are also various methods of determining the network throughput. In spite of these limitations, dynamic adaptive streaming methods currently dominate the video streaming market [27].

One of the most common methods is the ISO standard MPEG-DASH (Dynamic

Adaptive Streaming over HTTP) [27]. In this standard, the multimedia content is encoded into small segments with several different bit rates or resolutions and stored on an HTTP server. The actual multimedia files are accompanied by a Media Presentation Description (MPD). The MPD is a manifest of the available segments, their various bit rates, and potentially other relevant information [12].

Tiling

Tiling a large video into several smaller segments is a well explored video processing technique. The main idea in tiling is to divide the panorama picture horizontally and vertically into smaller regions that are encoded independently [38]. One of the common uses of tiling in this manner was Region of Interest (RoI) streaming. In RoI streaming, users select a specific RoI from a large, high quality video and the server streams only the data necessary for that specific RoI [34]. A common use case for this type of application might be an educational setting. The user would most likely focus on the professor or the blackboard. But the user would not change the ROI frequently, nor is there an unlimited amount off acceptable ROIs.

There are several previous works such as [2, 34, 38, 13] that attempt to optimize the dimension and number of tiled videos or how they are streamed. One common trade off is that spatially large tiles increase compression efficiency, but also tend to cause extraneous data to be streamed [38]. The exact number and dimension of the tiles presents an interesting optimization problem, that will be addressed more in the next section. However, the key concept in tiling is the idea that through tiling we can greatly reduce the total data we must stream. This can be accomplished in a variety of means.

In the Direct RoI Encoding method, only the tiles corresponding to the requested ROI are streamed at a high quality. The remaining tiles are not transmitted at all [2]. This can be a tremendous bandwidth saving for 360° videos because much of the scene is out of the users field of view. However, it is extremely susceptible to user movements. If the end user moves the RoI very quickly, they will move beyond the currently transmitted RoI and see black. In fact, if the total latency from user

movement to the scene being rendered is greater than 3 ms, humans may be able to perceive it [13]. This is clearly not attainable in current wired or wireless networks.

Tiling offers such obvious advantages that in 2015, a new amendment called Spatial Relationship Description (SRD) was added to the MPEG-DASH standard [12]. This SRD option provides the framework to define the spatial relationship between tiles. This provides one potential solution combining the benefits of tiling and DASH. However, in my experimentation several obstacles were immediately obvious. Using the MPEG-DASH SRD standard, each tile is played back at an independent resolution. This actually can be a very desirable characteristic, if the tiles in the center of the users field of view are played at higher resolution and others at less. But the standard provides no way to implement that and the tiles resolution were essentially random. The other major draw back was that each tile was played independently. They were in the right place, but frequently the edge between tiles was visible. I also frequently experienced a timing issue between the tiles where the tiles were not completely synchronized in time. When the tile edges crossed the background, this was not noticeable. But when movement was introduced, for example a person walked across the field of view, the timing mismatch became quite obvious.

Feasibility of Tiling

Creating tiles from an original video is straight forward and easily implemented in software. Even adding DASH support is relatively easy with several existing tools for creating the MPD file. Adding the SRD options is slightly more challenging because it is not widely implemented, but it is certainly not computationally expensive. Stitching the tiles back together is far more complex, especially when done in software. I ran a series of experiments to identify the computational difficulty of different tiling mechanisms. In all the experiments I streamed the same 360° video from an Amazon EC2 Cloud server to my quad core laptop. The first experiment, shown in Figure 3.1 did not utilize any tiling. Next I tested the MPEG-DASH with SRD option. I used nine tiles with four different bit rate representations for this

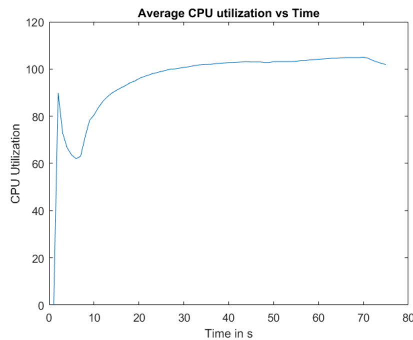

Figure 3.1: CPU utilization with no tiles.

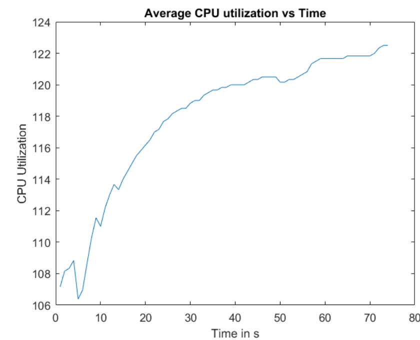

Figure 3.2: CPU utilization for DASH with SRD.

implementation. While this option does suffer from quality issues as discussed earlier, it offers very low computational complexity, as seen in Figure 3.2. I then ran three custom tiled experiments. For these, I created a tiled version of the video consisting of nine tiles. I then encoded each tile with two different qualities (bit rates), high and low. I then requested the nine tiles as independent streams, stitched them back together, and played the final video using FFMpeg and FFPlay. For the first set of experiments, I used only the low quality tiles. I then used only the high quality tiles. Finally, I requested five of the tiles at high quality and the remaining four tiles at low quality. The resulting CPU utilization followed a similar pattern for all three tests. Figure 3.3 shows the CPU utilization when all nine tiles were high definition. There is a very high initial spike which then drops down to a more constant CPU utilization. Figure 3.4 is the plot for the mix of high and low definition tiles. It follows the same pattern, but with slightly lower CPU utilization after the initial spike. The graph for the low tiles followed the same pattern, with even lower utilization after the initial spike. Table 3.1 summarizes the average CPU and memory utilization for each method tested. It is clear that no tiling and DASH are the most efficient for the end user device computationally. There is also a significant jump from the low quality tiles to the mix and the high quality tiles.

A few other observations are relevant. First, the experiments in this section were all conducted using the entire video. The end user would obviously not need all

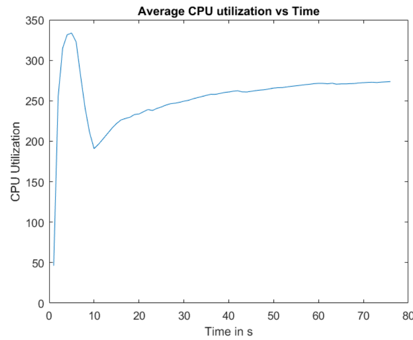

Figure 3.3: CPU utilization for 9 high definition tiles.

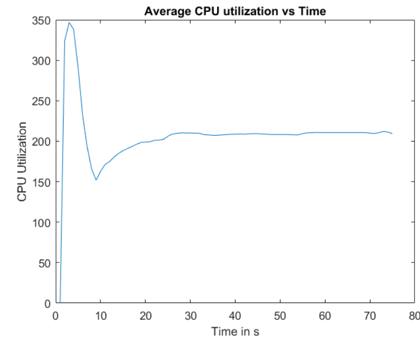

Figure 3.4: CPU utilization for 5 high and 4 low definition tiles.

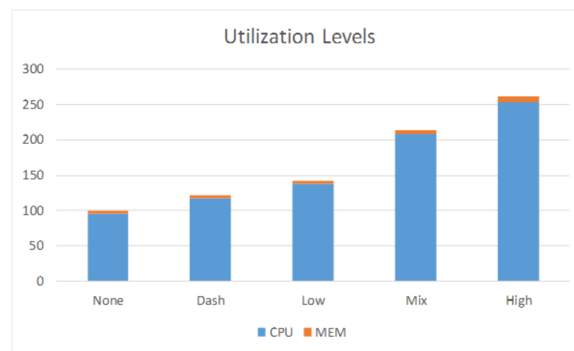

Table 3.1: Comparison of CPU Utilization for different tile schemes.

the tiles. Reducing the number of tiles that need to be stitched together improves performance on the user device. This trend would guide us to using larger tiles. But that minimizes the network bandwidth saving because most of the video ends up being transmitted. Second, using an end user device hardware to decode and stitch the tiles together would certainly be faster than doing it in software. However, most head mounted devices (especially phones) have limited hardware decoder and GPU capabilities. High GPU utilization also is far more expensive in terms of power consumption, making this option less attractive for portable devices. Despite these drawbacks, it seems that decoding the streams and stitching together the final video in hardware will offer the best performance. In the remaining sections, I focus on using hardware decoders and GPU to combine the separate tiles. This

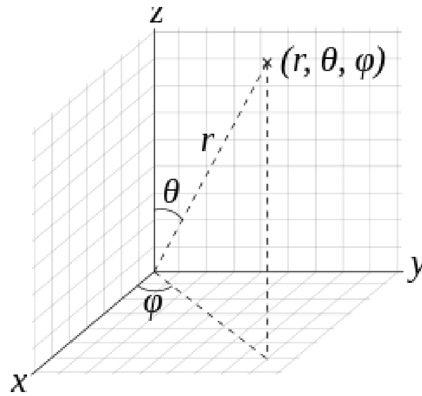

Figure 3.5: Spherical Coordinates.

choice is not necessarily the only option, but it seemed the most feasible given the CPU requirements for stitching and the extra delay introduced by the process.

Optimizing Number and Shape of Tiles

At first glance, the problem of optimizing the size and shape of the tiles seems strait forward. VR and 360° videos are stored in an equirectangular format; that is, they are stored flat with an aspect ratio of 2:1. The naive approach is to set up a simple optimization problem that fits a square field of view onto the equirectangular video and finds the smallest possible tile shape for every possible field of view (FOV). A limit could be set on the maximum number of tiles we are willing to stream. However, this straight forward approach dramatically fails.

VR and 360° videos are typically created using spherical coordinates. In spherical coordinates, every point is mapped using: r, the distance from the origin to the point; θ, the inclination or polar angle which ranges from 0 to π; and ϕ, the azimuth angle which ranges from 0 to 2π. Figure 3.5 demonstrates a point in polar coordinates. This is a logical and simple way to map the video to the recording source and later from the user back to the correct FOV in the movie. However, it is not possible to store or transmit videos in this way. So, VR and 360° videos are converted to equirectangular format by mapping the spherical azimuth angle, ϕ, to the x coordinate and the spherical inclination angle, θ, to the y coordinate. This is a

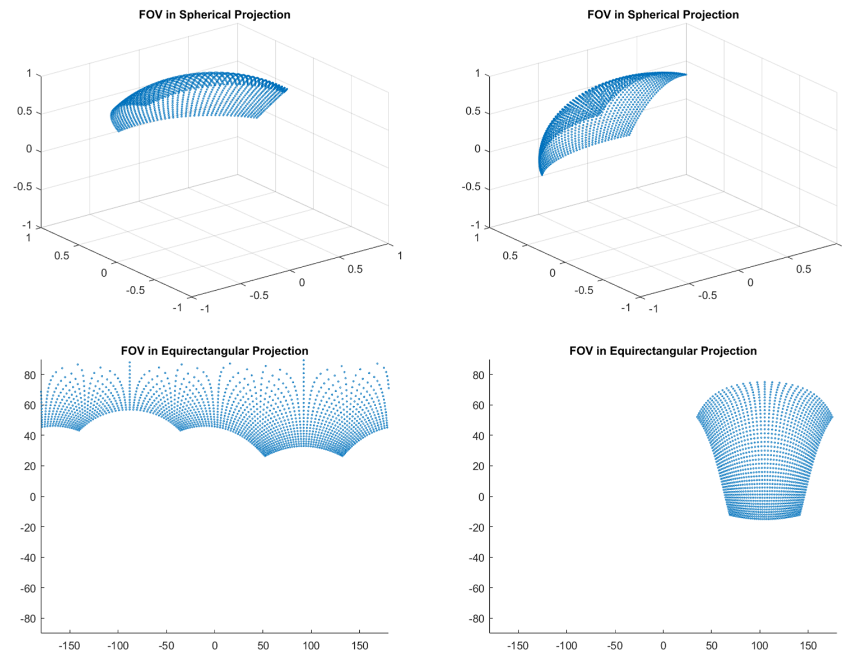

Figure 3.6: Spherical to Equirectangular Mapping.

computationally effective mapping and it is entirely invertible, both of which make it nearly ideal. However, it also leads to redundant information being stored near the poles and unusual mappings when users look toward the poles. This is the major obstacle in using the naive approach. To better understand and demonstrate this, I created a Matlab script which first mapped a 90° square field of view onto a sphere. I then transform that spherical FOV into an equirectangular FOV. Figure 3.6 is an example of two such FOV mappings. It is immediately obvious that as users look toward the pole, the entire horizontal axis of the equirectangular mapping is needed. Any attempt to optimize the number and shape of the tiles must take this effect into account.

There are a few other assumptions that must be made to effectively optimize the tiling. First, the actual FOV needed by the end user devices is critical. The majority of commercial VR devices have a field of view between 90° and 110° in both the vertical and horizontal directions. I optimized the tile size for a 90° field of view.

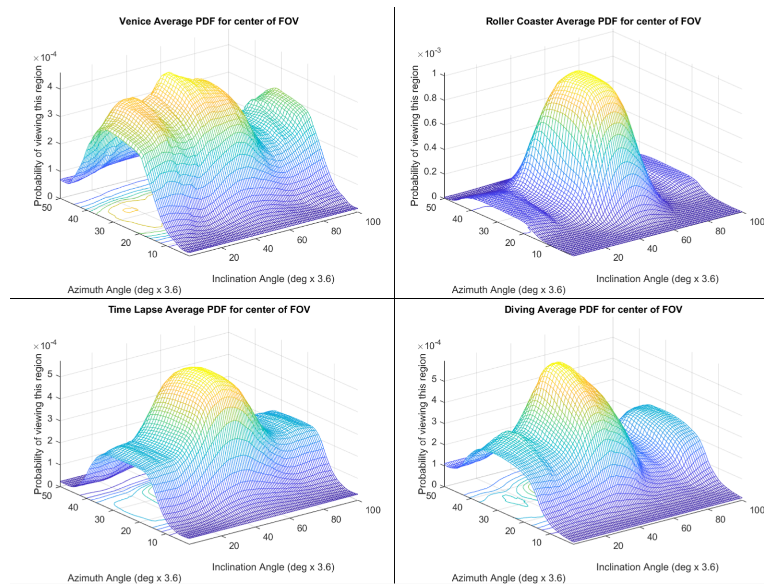

Figure 3.7: Average pdf for several movies.

The second major assumption is the maximum number of tiles the user device can decode. To this end, we looked at several cutting edge VR capable cell phones as platforms and how many streams they could decode in hardware. We found most modern phones offered at least 8 hardware decoders. To transmit VR videos, we actually need separate streams per eye, so that means we can utilize 4 hardware decoders per eye. Therefore, I used 4 as the maximum number of tiles we could transmit at any given time. The third and final assumption I made was that the center of the users FOV will follow a normal distribution for the inclination angle while the azimuth angle follows a uniform distribution. This last assumption does not affect the number or size of the tiles, but it does lead to a nice joint pdf in (θ, ϕ) to use in determining the expected number of tiles needed and expected bandwidth savings. This pdf is a good approximation of user head movements. The authors of [6] maintain a public data set of 63 users head movements while they watched 5 different 360° movies [1]. The average pdf for 4 of the movies can be seen in Figure 3.7. While the ideal model does not take into account the variations seen in the

[1] http://dash.ipv6.enstb.fr/headMovements/

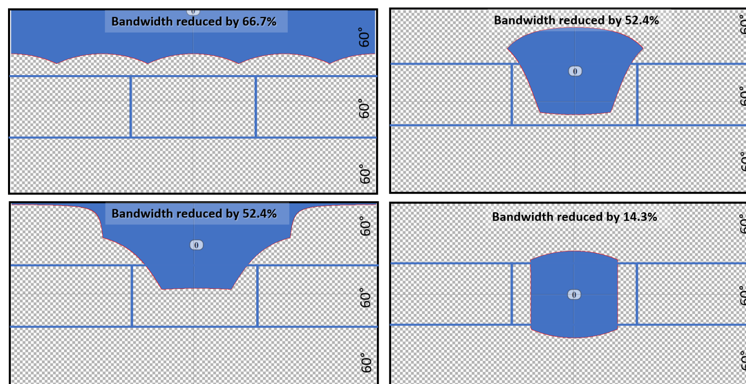

Figure 3.8: Option 1.

inclination angle, those variations do not affect the tiling schemes I proposed.

Based on these assumptions, I created several tiling options. I then used recursive testing to find and optimize the three best tiling options. I created a Matlab script to test each tiling option and calculate how many tiles are needed for every possible viewing angle. I discretized the possible viewing angles to 1° increments in both θ and ϕ. I calculated the number of tiles needed for every viewing angle and stored the number of tiles required in a matrix (180 x 360). I was then able to multiplied that matrix by the joint pdf at each of those discrete points to get an expected number of tiles at that point. Adding all the values of this final matrix gives the total expected number of tiles.

Option 1 consists of a single set of tiles unequally spaced. The equirectangular projection is split into three vertical segments of 60°. The center segment is split into three horizontal segments of 120° each. Figure 3.8 shows the tiles of Option 1 with a few representative FOVs. The number of tiles required for each viewing angle (the center point for the total FOV) are shown in in Figure 3.9. The expected number of tiles needed is 3.17, but the tiles are unequal size. I re-ran the simulation separating out the tiles based on their size to get a better idea of the bandwidth required. I used the same process as prior, but now I had two tile matrices for the two tile sizes. Option 1 will require, on average, 62.3% of the original bandwidth.

Option 2 consists of a single set of equally sized tiles. The equirectangular

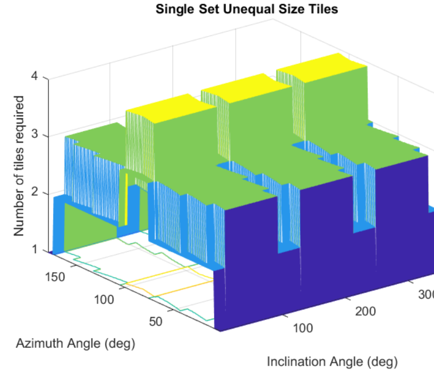

Figure 3.9: Option 1, Number of Tiles Required.

projection is split in half vertically and in thirds horizontally. Figure 3.10 shows the tile spacing with the best and worst case scenarios for the number of tiles needed. It also demonstrates the equivalent spherical field of view generated from the tiles that would be sent. The number of tiles required for each viewing angle are shown in Figure 3.11. It is immediately obvious that there are viewing angles that require 5 tiles. However, these angles account for only about 0.09% of the total angles. If we simply drop the fifth tile, we would lose 1-2° of the vertical FOV at the bottom or top of the frame. Because these shortcomings would be infrequent and have a minor impact on the user's experience, it may be acceptable to still use this scheme. The expected number of tiles required for Option 2 is 3.58 and on average this option should require 60.2% of the original bandwidth.

The third option consists of two separate sets of tiles. The first set of tiles would be used when the center of the FOV is near the "equator" ($\theta = 90°$). More formally, the equatorial tiles are used when the users FOV is entirely within 60° of the equator (i.e. the entire FOV satisfies $30° < \theta < 120°$). This set of tiles crops the top and bottom 30° from the equirectangular projection and then splits the remaining frame into 10 equally spaced strips. This can be seen on the left side of Figure 3.12. The second set of tiles are used when the center of the FOV is near either "pole". More formally this is anytime the FOV has an any inclination angle, θ, that is less than or equal to 30° or greater than or equal to 120°. This set of tiles splits the vertical

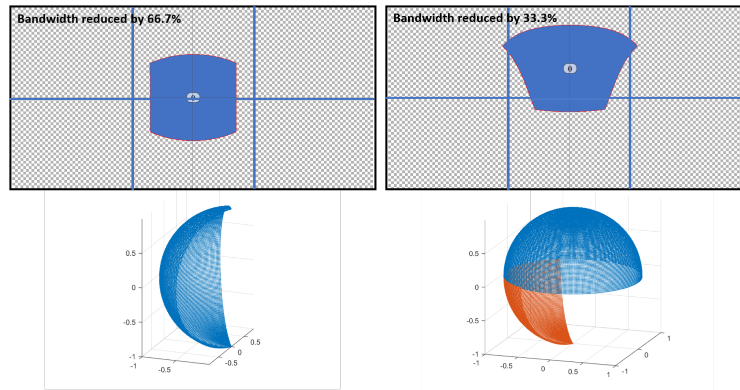

Figure 3.10: Option 2.

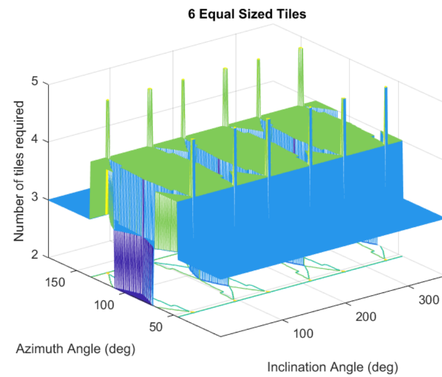

Figure 3.11: Option 2, Number of Tiles Required.

dimension into three 60° segments. I further divide the middle segment into 6 equally spaced horizontal segments. The "polar" tiles can be seen on the right side of Figure 3.12. Because there are two sets of tiles, I plotted the number of tiles separately in Figures 3.13 and 3.14. According to the joint pdf described previously, the equatorial tiles account for about 40% of the use cases. The expected number of tiles is 3.63 and the expected bandwidth needed is only 24.2% of the original. The polar tiles account for about 60% of the use cases. The expected number of tiles for the polar tiles is 3.64 which is about 48% of the original bandwidth. Combining these, the total expected bandwidth needed for Option 3 is 38.5% of the original bandwidth.

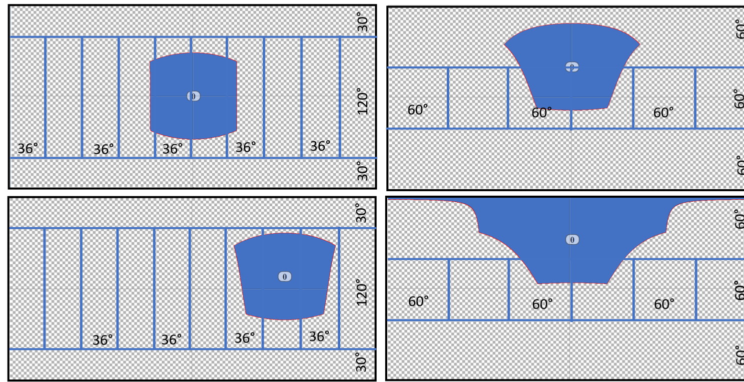

Figure 3.12: Option 3.

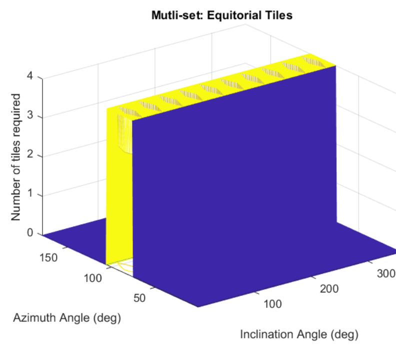

Figure 3.13: Option 3a, Number of Tiles Required for Equitorial Set.

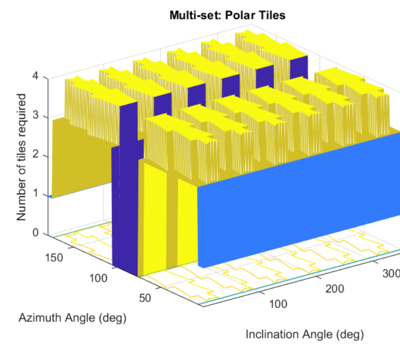

Figure 3.14: Option 3b, Number of Tiles Required for Polar set.

I also looked at the minimum bandwidth needed for an ideal oracle case. In this case, the server knows exactly what pixels the client needs at any given time and only sends those exact pixels from the original. Based on the ideal distribution discussed earlier, the expected bandwidth needed for ideal cropping is 14.4%.

It is quickly clear that we can expect to achieve significant reduction in the expected bandwidth required with all three options. Option 3 saves far more than Option 1 and 2, but at the expense of being more complicated. Option 2 saves slightly over option 1, but has a risk of losing some data. To verify these models, I used the public data set of head movement [2] to calculate the expected bandwidth

[2]http://dash.ipv6.enstb.fr/headMovements/

	Diving	Paris	Roller Coaster	Time Lapse	Venice
opt 1	56.45%	57.63%	57.58%	57.18%	56.80%
opt 2	59.01%	60.95%	60.47%	60.14%	59.36%
opt3	49.34%	49.99%	49.92%	49.78%	49.54%
ideal	15.25%	14.22%	14.59%	14.56%	15%

Table 3.2: Comparison of bandwidth utilization for different tile schemes based on 5 different videos.

for each tiling option based on the observed head movements for the 5 movies. The results can be seen in Table 3.2. Option 1 consistently uses slightly less bandwidth than the model I used. Option 2 is quite close to the model's predictions. Option 3 consistently uses about 10% more bandwidth than the model predicted. Option 3 still has the greatest bandwidth savings at just over 50%.

While all these options explore tiling the equirectangular frame itself, there are other options for storing and transmitting the video. Cube, pyramid, and dodecahedron mappings are three such option that removes much of the extra information near the poles and are fundamentally more compressed to start with [6]. However, they introduces other issues such as the needed tiles not being physically adjacent to each other when saved to a file. Some of the methods, such as the pyramid, are especially sensitive to head movements [6]. While the analysis I have provided here is not exhaustive, it does provide a clear path for analyzing tile schemes and their expected effect on bandwidth.

3.3 Improving VR Performance with Edge Computing

To overcome the high bandwidth requirements of streaming VR videos, I have shown that it is possible to tile the video and, based on the users FOV, send only the needed tiles to reduce the throughput by 50%. However, even with the number and size of tiles and the stitching method perfectly optimized, there is a major limitation based on network delay. Because the tiling options are not sending the

whole video, the user must request new tiles whenever they move their head. Total latency for VR systems from user movement to screen response should be 3ms to 10ms or less [6, 17]. Network round trip times, not including any video processing times by the server or client, vary dramatically based on both the physical distance and network topologies between the cloud server and user device. However, for major commercial servers they are frequently on the order of 40 to 90 ms. This latency is at least an order of magnitude greater than a VR video streaming system should have. One way to fix this issue is by moving the video storage location from a distant cloud server to the very edge of the network; to the Wi-Fi access point itself.

Wi-Fi Last Hop Latency

While it is obvious that moving the server closer to the end user will reduce latency, that does not guarantee the latency will be reduced below the goal of 3ms to 10ms. A thorough understanding of the latency between the Wi-Fi access point and the user device is needed. To understand this last hop latency, I conducted a simple series of experiments to test the round trip time from a Wi-Fi client to an access point. To do this I utilized ParaDrop, a cutting edge Wi-Fi access point with built in computational resources. ParaDrop is a research project out of the WiNGS Laboratory and the Computer Sciences department at The University of Wisconsin - Madison. Using this platform, I was able to install basic networking tools such as IPERF3 on the Wi-Fi access point itself. I utilized this platform as a local edge server, along with an Amazon Web Service EC2 as a distant cloud server to better define the latency benefits of moving to the edge in a typical home environment.

In the first set of experiments I had two end user devices connected to the ParaDrop Wi-Fi access point (AP). One device was used solely to introduce congestion to the network by downloading an IPERF3 stream from the AP. Before each test, I ran a short IPERF3 download test without any bandwidth limitations to determine the current maximum throughput. I then ran an IPERF3 download test with a specific percentage of the link as the target bandwidth. The other device ran a series

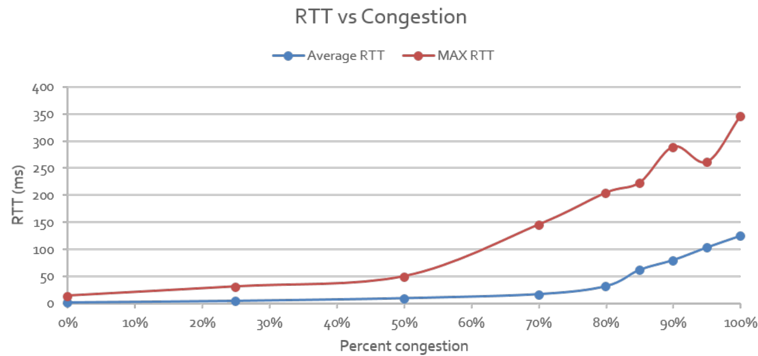

Figure 3.15: Round Trip Time to Wi-Fi AP vs Congestion.

of 100 ping tests to determine the round trip time (RTT) to the AP. The results can be seen in Figure 3.15. With no congestion, the mean RTT was 2 ms, while the most common RTT was 1 ms. The mean RTT increased in slowly in a linear manner until about the 70% congestion mark. At that point, the mean RTT rapidly increased. This gives us a very useful upper bound on the congestion load we should place on the wireless AP. As long as there is less than 70% congestion, we should expect round trip latency to the AP of less than 18 ms on average. I also conducted this experiment with the cloud server as the endpoint. There was a similar pattern in the relationship between congestion and latency. However, the 70% congestion cut off gives an average RTT of 125ms to the cloud server. This shows that moving the video server to the extreme edge offers an order of magnitude improvement in network latency.

To get a better understanding of the improvement offered by moving the video storage to the extreme edge, I ran a series of experiments comparing playback times for three video storage locations. For the first location, I saved the video to my laptop. For the second location, the video is stored on the ParaDrop AP. The video is stored on my Amazon Web Service EC2 server for the third location. I then created several simple bash scripts to compare different playback scenarios. I ran each script 10 times to each location, timing the execution time of each script. In this way, I could get a very clear idea of the extra time a user experiences by moving

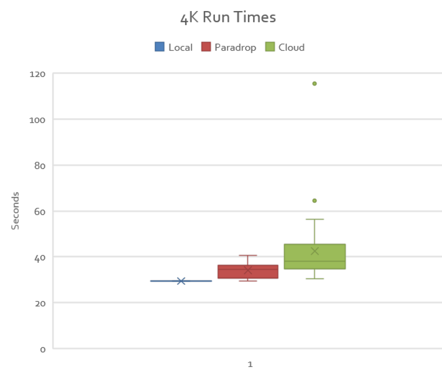

Figure 3.16: Run Times for 4K Video.

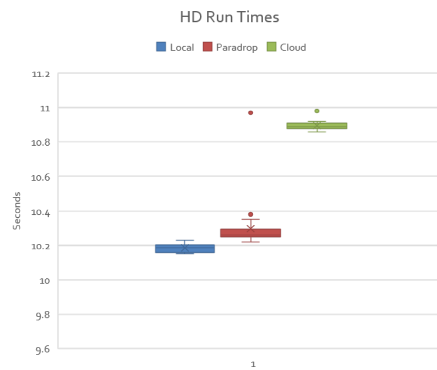

Figure 3.17: Run Times for HD Video.

the physical storage location of the video.

In the first scenario I played a 30s long clip of a 4K video. This video was chosen because it required a very high bandwidth to stream. In fact, the bandwidth was close to the IPERF3 tested maximum bandwidth to the Cloud server. The results can be seen in Figure 3.16. The playback time depended greatly on the instantaneous network performance. There was a noticeable degradation in the quality of the video when switching storage locations from local to ParaDrop and from ParaDrop to the Cloud. On average, streaming from ParaDrop added 4.7 seconds while streaming from the cloud added 13.1 seconds.

In the next scenario, I used a 10s clip of an HD 360° video. In this scenario, and all the following ones, the bandwidth needed to stream the video was well below the stable available bandwidth. In other words the network throughput was not a bottleneck. The results of this scenario are shown in Figure 3.17. There was no perceptual change in the quality of the video for any of the locations. On average, streaming from ParaDrop added 112 ms to the total execution time while streaming from the cloud added 712 ms.

For the third scenario, I used the same video from the second scenario. However, this time I added a simple pause command into the video playback. While the video playback was paused the video download also paused. The results of this

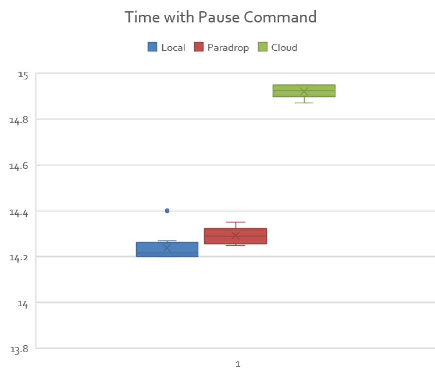

Figure 3.18: Run Times with Pause.

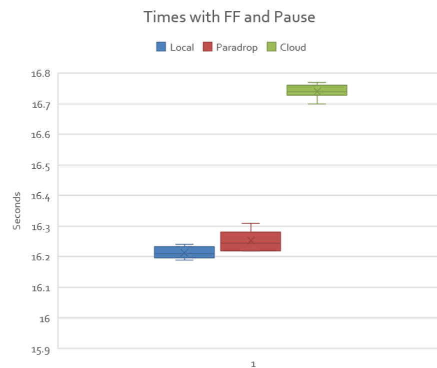

Figure 3.19: Run Times with Pause and Skip Forward.

experiment can be seen in 3.18. Adding the pause commands to the video did not add any latency. In fact, the average additional latency from the ParaDrop AP was only 55 ms, while the added latency from the cloud server was 683 ms.

Finally, for the fourth experiment I used a 30 second clip of the same video as the previous two experiments. This time I added both pause and skip forward commands. This test gives the best indication of how an interactive session will be affected by the location of the video server. The results can be seen in Figure 3.19. Again, there was no perceptual difference in quality between any of the server locations. On average, the ParaDrop router only added 40 ms of delay, while the cloud server added 527 ms of delay.

From all of these tests it is obvious that moving the video server to the extreme edge of the network offers greater than a 10x reduction in total latency. But the added latency, even when using the ParaDrop access point as the media server, is greater than the minimum network latency found previously. The most likely cause of the additional latency is the added time is due to initial connection establishment. This extra time occurs entirely before playback starts and would not affect the user experience.

Moving the video server to the extreme edge of the network clearly offers a huge reduction in latency. However, the added latency may still be greater than the

goal of 3ms to 10ms. A typical person can turn their head up to 120° per second [4]. However, most head movements occur at a much slower rate of about 60° per second [4]. One solution is simply padding each side of the required frame by some fixed spacial amount to ensure the user does not move their head faster than the server can respond with new tiles. Using the 70% congestion average RTT results from earlier, we can directly calculate how far a person can rotate their head in a single RTT (in degrees). Using the ParaDrop AP as the server would require 1.08-2.16° padding on all sides for a given frame. Using the cloud server would require adding 7.5-15° of padding on all sides. In other words, streaming from the extreme edge of the network, compared to a cloud server, reduces the padding needed from 683 pixels to 99 pixels per frame for a 1080p HD video. Moving the video server to the wireless AP both reduces latency and effectively reduces the bandwidth needed to ensure the same user video quality.

3.4 Practical Implementation

The results of the experiments covered in the previous sections are very promising. It is possible to tile the video into a reasonable (computationally feasible) amount of tiles and reduce the bandwidth needed by 40-50%. We can move the video's storage location to the very edge of the network and reduce the latency by an order of magnitude. Using these key ideas, I attempted to implement a working end to end test bed and demonstration. We utilized a Paradrop router running a Jetty Server instance as the server. We used a Google Pixel with Google Daydream as the user device. However, we ran into several obstacles that prevented a complete implementation. The rest of this section will briefly outline the obstacles we encountered.

The first thing that became clear was that to properly synchronize the tiles (especially when the tiles change) we need to segment the video in time as well as spatially. This problem is similar to the spatial size optimization problem. The longer the time chunks, the better compression we can obtain. But smaller time chunks are necessary for a more responsive system and to avoid sending unneces-

sary information. In this case I found a reasonable constraint; the system should not require more than one new tile in a given time chunk. If this were not the case, the system would either require too many hardware decoders or present too many tiles to the GPU or CPU to be computationally efficient. In general, a user cannot turn their head faster than 120° per second which is equivalent to 30° per 250ms. All of the proposed tile schemes have tiles with a minimum dimension of 30° or greater. This means that our time chunks should not be greater than 250ms; and to maximize compression they should be exactly 250ms. While this solves the problem of the chunk size, the client still needs to be able to sync a new tile to the existing tiles at any point within the time chunk. Currently, the maximum frame rate supported on most phones is 60fps. At this frame rate, each individual frame is displayed for 16.67ms. Therefore, I split each chunk into 10 sub-chunks of 12.5ms each. In DASH encoding, this simply ensures that there is a time stamp ever 12.5ms and the client can immediately search to that time stamp.

One technique we tried to implement was the DASH SRD option using the open source Android player, Exoplayer. As noted in section 3.2 there are timing issues between the tiles for this option. Neither Exoplayer nor the DASH specification provides a means of synchronizing the playback of the tiles. Small jitters in network delay can compound and become noticeable over time. Even worse, there is no means to synchronize a new tile when the user moves their head.

Another option I explored was to use FFmpeg to directly stream the tiles and stitch them together on the client. This is the method I utilized in section 3.2. While this is clearly possible on a laptop when using all tiles, there are two challenges to implementing this on a phone. First, it is computationally intensive. FFmpeg does support hardware acceleration for certain GPUs. It may be possible to write a custom android app that takes utilizes the phone's GPU to accelerate FFmpeg's re-encoding. However, there is a second problem with this method. It does not provide any way to sync a new tile to the existing ones. FFmpeg has a good system for syncing tiled videos based on the frame numbers. But it does not provide a way to quickly sync a new tile to a specific frame without downloading all the previous frames.

Next, we looked at creating a simple Android app that request the correct tile and passes the data stream directly to the phone's hardware decoders. In this way, we could ensure that each decoder corresponding to tile has a frame ready before we pass the output to the GPU's renderer. In this way we could guarantee there is no jitter between tiles. However, there are two problems with this strategy. First, we tried to use the DASH format to create small time chunks (250ms) from the original video. The individual DASH files are saved as ISOBMFF .m4s files. However, the hardware decoders are not able to directly extract the video data from the ISOBMFF files. We then tried to segment the video into small time chunks and save each one as a standard .mp4 movie file. But even increasing the chunk size to 400ms (and also increasing the sub-chunk size to 40ms) this nearly doubles the total file size for a video because it reduces the efficiency of the .mp4 compression. This has a net result of negligible reduction in bandwidth. Additionally, we found that while the decoders can play back a file arbitrarily fast, they take a noticeable amount of time to initialize and open a file.

Finally, we looked at using Exoplayer to request the tiles in DASH format, but then we wrote some custom code to hand the extracted video frame to a decoder. In this way, we hoped to get the best of both methods. However, every time Exoplayer handed a frame to the decoder, the phone simply turned off. In an effort to determine the issue, we used an Android emulator to run the same app. In this case, the program crashed and the decoder returned a 'Format not supported' error code. This line of attack needs further exploration.

Because of these various difficulties in rendering the final video, I leave the final system testing and demonstration for future work. In addition, all my experiments and analysis were focused on 2D 360° videos, not 3D 360° or true VR. After a successful 2D 360° rendering solution is implemented and studied, the system should be extended to 3D 360° and then to VR. Finally, more study is needed to optimize the edge server to allow live streaming with tiling on the fly and distribution to multiple users simultaneously.

4 CONCLUSION AND FUTURE WORK

In Chapter 2 I successfully demonstrated that transport layer protocols, specifically TCP, can easily be improved for bulk transfer of video data. I implemented a solution in the Linux Kernel optimized for production 4G-LTE networks which improves throughput without degrading user experienced delay. However, further study should be done to see if this is an optimal solution for current LTE networks. In addition large scale introduction of LTE-A has already began in the United States and across the world. Further study of live LTE-A networks needs to be conducted in order to understand and optimize bulk video transport over these networks.

In Chapter 3 I successfully demonstrated two key elements to a working VR streaming system. First, I outlined a method to optimize video tiling to reduce bandwidth without degrading the user's experience. I successfully implemented three tiling options and found that a bandwidth savings of 40-50% was achievable. Second, I demonstrated that moving the actual video storage to the extreme edge of the network can reduce the round trip latency by over an order of magnitude. However, I was not able to finish a complete end to end VR-streaming system. The Wi-Fi access point based server, implemented on a Para Drop router, is functioning and largely complete. I was able to successfully stream only the needed video tiles to a Google Pixel phone and found the bandwidth savings to be very close to the predicted values. I was not able to render these tiles into a single image and display them. This is not unique to my proposal, I have not found any working system that effectively implements tiled video streaming to a phone. Future work should focus on developing a mobile application that can accomplish three tasks: (i) concurrently download multiple video streams, (ii) decode each stream independently using hardware decoders, and (iii) pass the decoded data to the GPU to render the final image onto a 360° spherical projection.

As more and more devices connect to the Internet via wireless connections, we will see a huge range in network requirements. Some devices may require extremely low latency but use relatively low bandwidth, while others may require

large amounts of bandwidth but do not have low latency requirements. Some applications, such as VR videos, will require both high bandwidth and low latency. While wireless technologies will continue to improve and evolve, they will not be able to provide the same stability and reliability as wired networks. Because of these range of factors, the wireless communication world, both academic and industry, must utilize a whole system approach to optimize the behavior of specific applications. Adjusting the way applications work, such as optimizing for VR, can greatly improve the user's experience by cutting the bandwidth needed. Moving video storage from distant clouds to the extreme edge of networks can greatly reduce latency. Utilizing TCP congestion control algorithms that are tuned to the specific wireless medium's characteristics, or to the specific needs of the application, will provide huge benefits in both throughput and delay. While abstracting wireless communications into separate layers is fundamental to allow integration of many different approaches, engineers and programmes must not forget the whole systems approach.

REFERENCES

[1] Ian F. Akyildiz, David M. Gutierrez-Estevez, and Elias Chavarria Reyes. The evolution to 4G cellular systems: LTE-Advanced. *Physical Communication*, 3(4):217–244, 2010.

[2] Patrice Rondao Alface, Jean-françois Macq, and Nico Verzijp. Interactive Omnidirectional Video Delivery : A Bandwidth-Effective Approach. *Bell Labs Technical Journal*, 16(4):135–147, 2012.

[3] E Altman, K Avrachenkov, and C Barakat. A Stochastic Model for TCP with Stationary Random Losses. *IEEE/ACM Transactions on Networking*, 13(2):356–369, 2000.

[4] Ronald Azuma. *Predictive Tracking for Augmented Reality*. Phd, University of North Carolina at Chapel Hill, 1995.

[5] CISCO. Cisco Visual Networking Index: Forecast and Methodology, 2015-2020. 2016.

[6] Xavier Corbillon, Alisa Devlic, Gwendal Simon, and Jacob Chakareski. Viewport-Adaptive Navigable 360-Degree Video Delivery. In *MMSys*, Klagenfurt am Worthersee, 2016. ACM.

[7] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. PCC: Re-architecting Congestion Control for Consistent High Performance. *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 395–408, 2015.

[8] Daniel Genin and Tassos Nakassis. Modeling queuing dynamics of TCP: a simple model and its empirical validation. pages 1–6, 2011.

[9] Jim Gettys. Bufferbloat: Dark buffers in the Internet. *IEEE Internet Computing*, 15(3):1–15, 2011.

[10] Sangtae Ha and Injong Rhee. CUBIC : A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel*, 42(5):64–74, 2008.

[11] Sangtae Ha and Injong Rhee. Taming the elephants: New TCP slow start. *Computer Networks*, 55(9):2092–2110, 2011.

[12] Mohammad Hosseini and Viswanathan Swaminathan. Adaptive 360 VR Video Streaming : Divide and Conquer ! In *IEEE International Symposium on Multimedia*, San Jose, 2016.

[13] Kai-lung Hua, Rong Zhang, Mary Comer, and Ilya Pollak. Inter Frame Video Compression With Large Dictionaries of Tilings : Algorithms for Tiling Selection and Entropy Coding. *IEEE Transactions on Circuits and Systems for Video Techn*, 22(8):1136–1149, 2012.

[14] Junxian Huang, Feng Qian, Yihua Guo, Yuanyuan Zhou, Qiang Xu, âˆŁ Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. An In-depth Study of LTE: Effect of Network Protocol and Application Behavior on Performance. In *SIGCOMM'13*, pages 363–374, Hong Kong, 2013.

[15] ICT Data and Statistics Division. ICT Facts & Figures. Technical report, International Telecommuniation Union, Geneva, 2015.

[16] Krister Jacobsson. *Dynamic modeling of Internet congestion control*. 2008.

[17] Jason J Jerald. *Scene-Motion- and Latency-Perception Thresholds for Head-Mounted Displays by*. Phd, University of North Carolina at Chapel Hill, 2010.

[18] Haiqing Jiang, Yaogong Wang, Kyunghan Lee, and Injong Rhee. Tackling bufferbloat in 3G/4G networks. In *SIGCOMM'12*, pages 329–342, Helsinki, 2012.

[19] Hyungsoo Jung, Shin Gyu Kim, Heon Y. Yeom, Sooyong Kang, and Lavy Libman. Adaptive delay-based congestion control for high bandwidth-delay product networks. In *IEEE INFOCOM*, pages 2885–2893, Orlando, 2011.

[20] Teruo Kawamura, Yoshihisa Kishiyama, Yuichi Kakishima, Shinpei Yasukawa, Keisuke Saito, and Hidekazu Taoka. LTE-Advanced âŁ" Evolution of LTE âŁ" Radio Transmission Experiments. *NTT DOCOMO Technical Journal*, 14(2):22–36.

[21] Leong Wai Kay. The Case for A New Rate-based TCP Stack for Mobile Applications.

[22] Leong Wai Kay. *A Rate-based TCP Congestion Control Framework for Cellular Data Networks*. PhD thesis, National University of Singapore, 2014.

[23] Robert J Kilduff. *Analysis of Congestion Models for TCP Networks by Masters of Engineering Science*. Masters of engineering, National University of Ireland, 2003.

[24] Vengatanathan Krishnamoorthi, Anirban Mahanti, Niklas Carlsson, Derek Eager, and Nahid Shahmehri. Quality-adaptive Prefetching for Interactive Branched Video using HTTP-based Adaptive Streaming. In *ACM International Conference on Multimedia*, number Mm, pages 317–326, Orlando, 2014.

[25] Franck Le, Erich Nahum, Vasilis Pappas, Maroun Touma, and Dinesh Verma. Experiences Deploying a Transparent Split TCP Middlebox and the Implications for NFV. In *HotMiddlebox'15*, pages 31–36, London, 2015.

[26] Brett Levasseur, Mark Claypool, and Robert Kinicki. A TCP CUBIC Implementation in ns-3 âˆ—. *Proceedings of the 2014 Workshop on ns-3 - WNS3 '14*, pages 1–8, 2014.

[27] Seong Yong Lim, Joo Myoung Seok, Jeongil Seo, and Tag Gon Kim. Tiled panoramic video transmission system based on. In *IEEE International Conference on Information and COmmunication Technology Convergence*, pages 719–721, 2015.

[28] Feng Lu, Hao Du, Ankur Jain, Geoffrey M Voelker, Alex C Snoeren, and Andreas Terzis. CQIC : Revisiting Cross-Layer Congestion Control for Cellular Networks. In *HotMobile'15*, Santa Fe, 2015.

[29] NGMN Alliance. 5G White Paper. pages 1–125, 2015.

[30] Binh Nguyen, Arijit Banerjee, Vijay Gopalakrishnan, and Sneha Kasera. Towards Understanding TCP Performance on LTE / EPC Mobile Networks. In *AllThingsCellular'14*, Chicago, 2014.

[31] Wouter Pasman and Frederik W Jansen. Latency layered rendering for mobile augmented reality. Number June, 2000.

[32] Qualcomm. LTE Advanced âŁ" Evolving and expanding in to new frontiers. *Https://Www.Qualcomm.Com*, (March):1–42, 2014.

[33] Hemant Kumar Rath, Anirudha Sahoo, and Abhay Karandikar. A Cross Layer Congestion Control Algorithm in Wireless Networks for TCP Reno-2. 2005.

[34] Guntur Ravindra and Wei Tsang Ooi. On Tile Assignment for Region-of-Interest Video Streaming in a Wireless LAN. In *NOSSDAV*, Toronoto, 2012.

[35] A. Roessler and M. Kottkamp. LTE- Advanced (3GPP Rel.11) Technology Introduction. *White Paper*, pages 1–38, 2013.

[36] Rohde & Schwarz. LTE Transmission Modes and Beamforming White Paper. 2015.

[37] Charalampos (Babis) Samios and Mary K. Vernon. Modeling the throughput of TCP Vegas. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31, page 71, San Diego, 2003.

[38] Y Sánchez, R Skupin, T Schierl, and Fraunhofer Hhi. COMPRESSED DOMAIN VIDEO PROCESSING FOR TILE BASED PANORAMIC STREAMING USING HEVC. In *IEEE International Conference on Image Processing*, pages 2244–2248, Quebec City, 2015.

[39] Narges Shojaedin. TCP-Aware Scheduling in LTE Networks. In *World of Wireless, Mobile and Multimedia Networks*, Sydney, 2014.

[40] Telecommunications Industry Association. 5G Operator Survey (2015). 2015.

[41] Telesystem Innovations Inc. Telesystem Innovations LTE in a Nutshell :. Technical report, Toronoto, 2010.

[42] V A Vasenin and G I Simonova. Mathematical Models of Traffic Control in Internet: New Approaches Based of TCP/AQM Schemes. *Automation and Remote Control*, 66(8):1274–1286, 2005.

[43] Ren Wang Ren Wang, G. Pau, K. Yamada, M.Y. Sanadidi, and M. Gerla. TCP startup performance in large bandwidth networks. *Ieee Infocom 2004*, 2, 2004.

[44] Keith Winstein, Anirudh Sivaraman, Hari Balakrishnan, and M I T Csail. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks Cellular networks are variable Verizon LTE uplink throughput. In *10th USENIX Symposium on Networked Systems Design and Implementation*, Chicago, 2013.

[45] Xiufeng Xie and Xinyu Zhang. piStream : Physical Layer Informed Adaptive Video Streaming Over LTE. In *MobiCom'15*, Paris, 2015.

[46] Qiang Xu, Sanjeev Mehrotra, Zhuoqing Mao, and Jin Li. PROTEUS: Network Performance Forecast for Real-Time, Interactive Mobile Applications. *Proceeding of the 11th annual international conference on Mobile systems, applications, and services - MobiSys '13*, page 347, 2013.

[47] Yin Xu, Zixiao Wang, WaiKay Kay Leong, and Ben Leong. An End-to-End Measurement Study of Modern Cellular Data Networks. In *Passive and Active Measurement Conference '14*, volume 8362, pages 34–45, Los Angeles, 2014.

[48] Yasir Zaki, Thomas Pötsch, Jay Chen, and Carmelita Görg. Adaptive Congestion Control for Unpredictable Cellular Networks. In *SIGCOMM'15*, pages 509–522, London, 2015.