Exploring P4 and its applications for QoS in video streaming

Matthew Bulger
School of Computing and Augmented Intelligence
Arizona State University
Tempe, USA
mbulger1@asu.edu

***Abstract*—P4 is a programming language for software defined network devices, and is a flexible system that can be used to engineer traffic in ways that were previously unattainable in the world of traditional, vertically-integrated forwarding and routing devices, particularly with regard to attaining Quality of Service in the context of video streaming applications. Using the FABRIC testbed as a platform for experimentation, my P4 program is able to separate video streaming traffic from unrelated cross-traffic, producing a statistically significantly increase in the average bitrate of video playback. I also explore the process of learning the P4 language through a variety of pre-built exercises, including the pitfalls that arise when transferring this theoretical knowledge to a practical setting.**

***Index Terms*—P4, QoS, video streaming, HTTP/2, QUIC, FABRIC testbed**

I. Introduction and Motivation

One of the foundational principles of the Internet's design is its best-effort packet delivery service model. In order to accommodate an arbitrary number of users, this service model trades off the ability to reserve bandwidth. In a world where the majority of activity consisted of spiky bursts of traffic, requesting relatively small files, this trade off was acceptable, and in fact, a large factor in its unparalleled growth. However, the nature of Internet traffic has already changed drastically; Cisco projected 82% of global Internet traffic is for streaming video content [1], an application which requires a guaranteed amount of throughput for an acceptable user experience. Protocols such as DASH exist to buffer and adapt the quality of a streamed video given the network conditions, but we can do better.

The advent of Software-Defined Networking has lead to a revolution in the field of networking-no longer bound by the proprietary, vertically integrated equipment of the past, technologies like P4 allow us to perform arbitrary parsing and manipulation of packets in order to flexibly provide a higher degree of network services at the core. It is the goal of this project to explore the use of P4 in providing better Quality-of-Service (QoS), particularly as it applies to video streaming.

II. Related Work

In a comprehensive survey on the uses of P4, Kfoury *et al.* [2] describe two distinct approaches for providing better QoS. Bhat *et al.* [3] use the Chameleon testbed to demonstrate a P4 program which extracts the *Stream ID* from the *HTTP/2* header, allowing the switch to prioritize the QoS-sensitive video streaming traffic along the "fast path", while falling back to a slower link for other forms of traffic. Using this relatively simple technique, Bhat *et al.* were able to achieve significantly higher average bitrate, as well as a reduction in the number of times the client changes from one bitrate to another (frequent quality changes can be quite distracting to the end user).

HTTP/2 is still the dominant protocol for video streaming, but due to its reliance on TCP (which is baked into operating systems, and thus slow to change), its performance improvements have stagnated over time. QUIC (which instead runs over UDP) is a new protocol that promises increased performance, and rapid development. Bhat *et al.* make brief mention of QUIC, and how their approach could be extended to provide the same QoS improvements running over the QUIC transport protocol, but the majority of their work, and associated reproducibility artifacts, were focused on running over HTTP/2. In my project, I attempt to extend this prior work to recognize QUIC flows as well. The final goal was to obtain an objective comparison in the performance improvement of using simply HTTP/2 versus QUIC, as well as with/without P4-enabled QoS enhancements. I used some of Arisu & Begen's work on QUIC in providing QoE (particularly their client DASH application) [4], [5] in my attempts to achieve this goal, but time constraints ultimately limited the results I was able to collect on this front.

III. Experimental Set-up

This project involves four distinct phases of exploration:

A. P4 Learning Exercises

The first phase of this project was primarily exploratory. In order to gain a familiarity with the P4 language, and how to deploy it on a software switch, I used Dr. Crichigno's platform of P4 exercises [6]. While these are primarily focused upon the security aspects of P4, these exercises provided me with a foundation of skills that was instrumental in implementing my later P4 program(s).

B. Simple P4 Network

Having gained a reasonable level of theoretical experience with P4, next, I deployed an extremely simple network topology on the FABRIC testbed, consisting of two hosts on

different subnets, connected with a single P4 switch acting as a router between the two. As the saying goes, "we must learn to walk before we can run", and thus, the objective is not to stream video between the hosts, but rather to simply communicate between them with ICMP Echo Requests/Replies. This demonstrates the ability of P4 to act as a standard L3 router, which was a solid foundation to build upon for further stages of the project.

Currently, the FABRIC testbed does not host any hardware P4 switches. Instead, a software P4 emulator will be used, namely BMv2 [7], which can be deployed on a Linux-based FABRIC node within the slice.

C. HTTP/2 Video Streaming

The next phase of this project revolved around exploring and extending the work of Bhat *et al.* [3] to create a P4 program that can distinguish between and prioritize HTTP/2 video streaming traffic, as opposed to other, less performance sensitive applications.

For the video streaming server application, the Apache 2 [8] HTTP/2 server was used, and for the client, the AStream [9] DASH client was employed. In order to test the ability to separate two different flows of traffic, I of course need a separate client and server to generate arbitrary traffic en masse; for this purpose, *iperf3* can be used. While *iperf3* is a tool primarily designed for measuring the average throughput of a link, it does so by sending large quantities of packets at a time, up to a maximum throughput, which can be specified using one of its parameters. First, an *iperf3* server process with default parameters is started on the server node. Meanwhile, the following command starts the *iperf3* client process on the client node:

```
iperf3 -c server_ip -t 130 -i 5 -u -b maximum_throughput
```

The *server_ip* is the IP address of the server, and the *maximum_throughput* is a value close to saturating the link. This client process will send as many packets as possible (up to the maximum throughput we input), and uses the responses from the server to measure the throughput of the connection. Since we are running *iperf3* in UDP mode, if the cross-traffic and video streaming traffic are routed across the same link, *iperf3* will eat up most of the free bandwidth, causing the bitrate of the video stream to suffer. If we split the cross-traffic and the video traffic across two different links, though (which I will call the "both paths" configuration), the cross-traffic and video stream do not need to compete, allowing the video stream to achieve a higher average bitrate.

D. QUIC Video Streaming

In the previous phase of the project, the DASH video stream was served over HTTP/2, which uses TCP as its transport protocol. These protocols are in widespread use across the Internet today, and the software to implement them is quite mature and easy to set up (in this case, Apache was used as the server application). However, HTTP/2 video streaming is on its way out, to be replaced by QUIC. While the QUIC protocol has only recently been solidified by the IETF, Google already uses it for every connection to their servers made by Google Chrome clients. It is clear that QUIC (and HTTP/3) will soon become the primary standard for streaming video content, thanks to its supposed increased flexibility and performance.

For this phase of the project, I attempted to replace my HTTP/2 based video server with one that streams video over HTTP/3 and QUIC. There are several options for server applications to use, but seeing as QUIC was primarily developed by Google, a logical place to start was with their *quic_server* [10] included in the Chromium source code. Another option I experimented with is Caddy [11], which provides a much simpler installation process, and takes care of many of the hassles that are present in Chromium solution, such as automatically installing the self-signed certificates required by TLS, a security measure that was optional in previous implementations of HTTP, but is now a requirement of HTTP/3.

Of course, communication with a QUIC server also requires a client application that can request segments using the same QUIC protocol. For this purpose, I experimented with Arisu & Begen's fork of AStream, which was adapted to run over QUIC [4], [5]. Since I used the original version of AStream in the previous phase of this project, this would allow me to collect the same metrics using the same "basic" adaptation method, limiting the altered variables to the protocol and server application used.

IV. Results

A. Exploration Results

The first phase of this project was extremely useful in getting me familiar with the various aspects of the P4 language. The exercises began very simply; they first introduce Mininet, a platform for instantaneously creating a virtual network with an arbitrary number of hosts and switches between them. Unlike the testbeds we have explored throughout the course, such as FABRIC, GENI, or SLICES, Mininet, is an entirely software-based solution for creating virtual networks. While the mentioned testbeds also implement a virtual network with the topology of your choosing, these nodes are physically deployed on bare-metal servers throughout the country (and even throughout the world), allowing the collection of data that more closely matches figures that would be seen in "real-world" deployments. Of course, with the "real-world" comes an unbelievable amount of complexity, and as such, starting my foray into the world of P4 and Software-Defined Networking was far simplified by starting with a simplified approach through Mininet.

While learning how to create my own P4 program was a priority, some of the greatest lessons I learned through these initial exercises was in the workflow for debugging and executing P4 programs. A P4 program itself solely defines the data plane behavior for a software defined switch, but without a control plane, the data plane is useless.

A P4 program defines the necessary tables, match conditions (longest prefix match vs exact match), and actions (forward

the packet, drop the packet, etc.) in a high-level language. Using a P4 compiler (the most common of which being P4C [12]), a P4 program is compiled into a JSON file, which can be executed on any P4 switch, implemented in hardware, such as an Intel Tofino switch, or emulated in software, using the P4 Behavioral Model v2 (BMv2), which can be run on any Linux node (albeit at significantly reduced performance). Given the cost associated with Tofino switches, the remainder of the paper will assume the usage of BMv2 as the implementation of P4 [7]. In either scenario, the P4 program is generic, and while it defines the structure of the tables that the switch will use, we still need a mechanism for filling the tables we define. Installing BMv2 provides a program entitled *simple_switch_CLI*[1]. Once the P4 data plane program is executing on the switch, this program can be used to add entries to any of the defined tables via the *table_add* command. Despite being one of the most important aspects of using BMv2, I found the documentation regarding the syntax of these commands to be extremely lacking, which is why having examples defined in the form of these P4 exercises was extremely useful.

B. Initial FABRIC and P4 Experimentation

With the solid foundation in the workflow used for developing, testing, and deploying a P4 application, and having encountered very few issues in completing the exercises, I felt confident in my ability to apply these skills to a physical testbed, as opposed to simply a virtual Mininet topology. I turned my attention to the FABRIC testbed, which similarly allows you to create an arbitrary network topology, but, unlike the exercises I had completed up to this point, and given that it is still quite an immature platform, FABRIC does not have a sole focus on P4, and required an extensive amount of work to set up the nodes and software P4 switches running BMv2.

To begin, I took the naïve approach of creating a Jupyter notebook from scratch, with the straightforward goal of learning how to install and instantiate a P4 switch, and communicate between two hosts across different subnets, as a standard router would be able to. Despite the humble goal, this task involved many more steps than planned.

Unlike most consumer-oriented software, due to the experimental nature of BMv2, and P4 in general, there is not a single, self-contained package nor command that installs all the necessary software, and a significant amount of time was spent installing the specific versions of necessary dependencies, such as *Apache Thrift*, *Nanomsg*, *Google Protobuf*, and BMv2 itself. Installing the dependencies from source, as the documentation suggests, is a time-consuming process.

The end hosts also required a minimal amount of configuration; I set up *juliet* to be on the $192.168.1.0/24$ subnet (with IP address $192.168.1.2$), and *romeo* to be on the $192.168.0.0/24$ subnet (with IP address $192.168.0.2$). See Figure 1 for the network topology used in testing.

[1] *simple_switch_CLI* has been succeeded by *runtime_CLI* within the BMV2 [7] package, but in general, their syntax and usage is similar enough that their minute differences do not matter in the context of this paper.

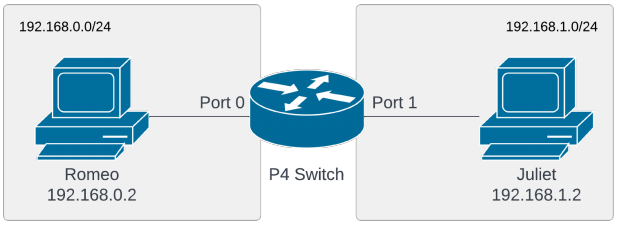

Fig. 1. The first attempt at implementing a network using a P4 switch involved a simple network topology with two hosts, *romeo* and *juliet*, with a single P4 switch connecting them. For this initial test, all nodes were deployed on a single site, namely *TACC*.

With everything configured, I attempted to use the aptly-named $basic.p4$ program, which implements basic Layer 3 routing (specifically the IPv4 protocol) to test out my network configuration. Similar to those provided in the previously mentioned P4 exercises, I used *simple_switch_CLI* to populate the routing table with simple rules to route traffic destined to the respective subnets of my end hosts, *romeo* and *juliet*, using the following commands:

```
table_add ipv4_lpm ipv4_forward 192.168.0.0/24 => 0
table_add ipv4_lpm ipv4_forward 192.168.1.0/24 => 1
```

The first command inserts an entry into the *ipv4_lpm* table, matching packets destined to the $192.168.0.0/24$ subnet, which will then take action *ipv4_forward*, egressing the incoming packet to port 0.

To test the communication between the hosts, I used the *ping* command from both hosts. Using the logs generated by BMv2, I noticed that the packets correctly arrived at the P4 switch, and all headers were processed correcting, including the L2 Ethernet header and L3 IPv4 header, and the destination IP address was correctly matched to one of the two inserted rules in the table. However, while the switch logged that it was "egressing packet on port X", where X was either port 0 or 1 depending on the host sending the packet, no matter what I did, the packet would never arrive at the next hop (in this case, the recipient host). The lack of logs or detailed output from BMv2 made this quite a difficult issue to debug.

In order to gain some perspective, I decided to take a further step back to eliminate some variables from the equation. Rather than using a slice where I installed BMv2 manually, I moved to one of the provided example FABRIC notebooks. Instead of installing everything from source, this notebook used a prebuilt Docker image containing all necessary dependencies, ensuring I did not miss any steps of the processes. However, instead of using $basic.p4$ as before, this notebook used $basic\_tunnel.p4$, an extended version which replaces the standard IPv4 protocol with an extremely simple custom protocol that uses an exact integer to identify the recipient of a given packet (as opposed to an IPv4 address and longest prefix matching). Despite being one of the pre-built examples for the usage of P4 within the FABRIC testbed, this too was

not "plug-and-play"; the provided scripts required numerous modifications to properly configure the network interfaces of each of the switches[2]. Surprisingly, this worked exactly as expected; I was able to send packets from one host to another. In fact, despite the topology of this network consisting of two hops between each end host, rather than the single switch as in my previous unsuccessful, I was still able to communicate between the nodes without issue.

Heartened by my newfound success, I was confident in my ability to switch from the $basic\_tunnel.p4$ program, back to the original, more simplistic $basic.p4$ program, which uses the standard IPv4 protocol in layer 3. However, upon making this simple modification, I found myself back at square one; the packets would successfully arrive at the first hop, and, according to the logs, be egressed to the appropriate output port, but the packets refused to arrive at the next-hop node.

C. Address Resolution Protocol

Ultimately, after considerable debugging using *tcpdump*, in conjunction with runtime P4 log files, the culprit behind the inability to communicate between hosts was narrowed down to ARP. ARP, or "Address Resolution Protocol", is responsible for mapping layer 3 IP addresses to physical layer 2 MAC addresses. The goal is to identify the layer 2 address of the next-hop router, who will then re-examine the destination IP address contained within the packet in order to determine the MAC address of the next hop, and so on, until the packet arrives at a router who lies on the same subnet as the destination host, at which point the packet can be directly sent to the recipient.

Typically, when the corresponding MAC address does not exist in the ARP cache for a given IP address, the host will broadcast an ARP request across the subnet, which any attached routers will receive. If the router finds that it has a matching route to that destination, it will respond with an ARP reply, containing the MAC address of *its own interface that received the ARP request*; the router is not guaranteed to be on the same subnet as the destination, it simply knows a route of the next-hop to that destination. As such, *it does not respond with the MAC address of the destination itself.*

Due to the relatively simple P4 programs this work is based upon, the P4 switches used in this network are not able to reply to any ARP requests they may receive. As such, this configuration requires the ARP caches, both on the end hosts, and the routers themselves, must be statically configured, which lead to my crucial mistake. Rather than configuring ARP caches with the MAC addresses of the next hop, they were configured with the MAC address of the final destination. Thus, when a packet was sent to the first hop router, the provided MAC address did not match that of the router's input network interface, and the packets were ultimately discarded,

[2]Due to a seeming bug in the FABLIB API, the provided setup script for the P4 switches was unable to identify the correct names of the switches' network interfaces, requiring them to be manually specified when running the Docker image. Further modifications were made to enable logging on the switches in an attempt to debug the issues I faced in transmitting packets.

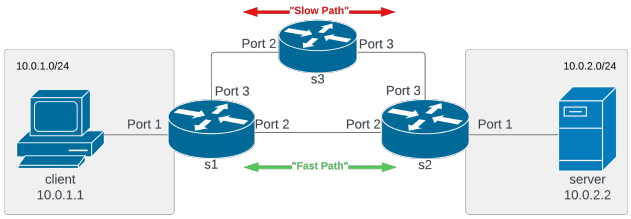

Fig. 2. The topology used for testing HTTP/2 video streaming consists of a client and server node at the edge of the network, directly connected to P4 switches *s1* and *s2* respectively. The "fast path" for packets is a direct link between *s1* and *s2*, with no bandwidth restrictions imposed. The secondary "slow path" between the switches contain another intermediate hop at the third P4 switch, *s3*. This switch's interfaces have a bandwidth cap of 5 Mb/s.

which explains why they never arrived at the next-hop router. Even once the ARP caches were configured properly on the end hosts, since the routing table did not contain the proper MAC address of the next hop, the packets were still dropped. Once both the routing tables and ARP caches were configured correctly, everything worked flawlessly, and I was finally able to perform a ping between the hosts.

It is not the goal of this paper to provide excuses for such a mistake, but I would note that a contributing factor was, counterintuitively, the relatively simple topology used for testing. The end hosts were directly connected to the P4 switches, without any layer 2 switching devices between them, as would be expected in a more realistic application. It is easy to forget that, despite the relative "closeness" of the end hosts (only two hops away), they are on entirely different subnets, and are totally unaware of each other's layer 2 addresses—after all, if they were, then there would be no purpose for having layer 3 addressing! Had there been an additional L2 hop on any of the subnets, this distinction would have been more obvious, as the MAC address of the next hop would be needed by this switch.

D. HTTP/2 Video Streaming Results

Having demonstrated the successful communication between the hosts using P4 switches, we are finally able to extend the simple P4 routing program to provide differential QoS to HTTP/2 video streaming traffic, similar to that done by Bhat *et al.* [3]. Figure 2 details the topology used for experimentation.

Ultimately, the commands of my new P4 program were of the following format (using commands from *s1* for reference):

table_add ipv4_lpm set_nhop 10.0.1.0/24 => [h1 MAC] 1 [h1 MAC] 1
table_add ipv4_lpm set_nhop 10.0.2.0/24 => [s2 MAC] 2 [s3 MAC] 3

Clearly, the longest prefix match on the IP addresses are the same as before, but I have added new fields to the action. The first MAC address and integer represent the MAC address and egress port for "fast path" traffic (i.e. if traffic is identified as video streaming traffic, it will be transmitted across this link). Likewise, the second MAC address and integer represent the

MAC address and egress port for "slow path" traffic (i.e. if the traffic is not video streaming traffic, it will be transmitted across this link). In cases were there is only a single link connecting the nodes (such as between *h1* and *s1*), we use the same MAC address and egress port combination for both, such that all traffic is transmitted across the same link, regardless of how the traffic is identified. By altering the combination of destinations for video streaming traffic and cross-traffic, I was able to control whether all traffic was transmitted across a single link (as would be typical in standard L3 routing), or whether the two traffic types should be transmitted across separate links.

In this proof of concept, the mechanism for differentiating video streams from cross-traffic is admittedly rudimentary. Bhat *et al.* use the *StreamID* header of HTTP/2 for this purpose [3], but I instead simply use the TCP port for this—since only video streaming traffic will use port 80, while the cross traffic uses port 5201, this was sufficient in differentiating the traffic for my simple example. However, in a real-world scenario, all HTTP/2 traffic uses 80, not just video streams; reverting to the *StreamID* would remove this limitation.

Having creating a working P4 program that can route traffic across two different links, I used the following methodology for collecting quantitative data on its performance. The first step involved installing the client and server software on two distinct nodes [8], [9]. On the server, the "Big Buck Bunny" dataset [13] was used, and encoded into four-second segments of differing bitrates to be requested by the client. For all test runs, the "basic" adaptation policy was used to fetch and play back a total of 30 segments. Given that, by default, each of the links operate at approximately the same speed, an artificial bandwidth cap was placed on switch *s3* of 5 Mb/s. Furthermore, for consistency, and to ensure that each run is able to saturate the provided link, we also cap the bandwidth of the server to a slightly higher 6 Mb/s. For each trial, traffic was routed according to one of three different configurations: "Both Paths" (all cross-traffic is transmitted on the slow path, while all video traffic is transmitted on the fast path), "Fast Path" (all traffic transmitted on fast path, while the slow path goes unused), or "Slow Path" (all traffic transmitted on slow path, while the fast path goes unused). At least three trials were executed for each routing configuration. In all cases, the P4 programming running on the switch is identical; the only difference between the configurations are the entries inserted into the *ipv4_lpm* table of each of the switches.

As shown in Figure 3, for around the first 30 seconds of each trial, the DASH client steadily increases the bitrate it is requesting, and since none of the configurations saturate the link at this point, the video quality is approximately the same for each. However, as time goes on, we see each of the configurations reach a different maximum bitrate. We can clearly see that when the cross-traffic is separated from the video streaming traffic, this maximum bitrate is significantly higher than when all traffic is transmitted across solely the fast or slow links. If we take the average bitrate across all trials for a given configuration, as we see in Figure 4, it is clear

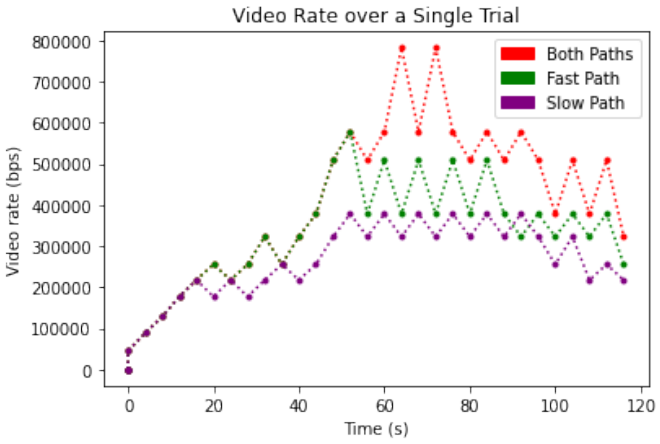

Fig. 3. Each line represents the current bitrate of the video playback over the course of a single trial. In the "Both Paths" runs (red), the P4 switches transmitted all the video traffic over the "fast path", while all the cross-traffic was sent over the "slow path". In the "Fast Path" runs (green), all traffic was transmitted over the faster link, while the slow path went unused. Likewise, the "Slow Path" configuration (blue) transmits all packets across the slower link, while the faster link went unused.

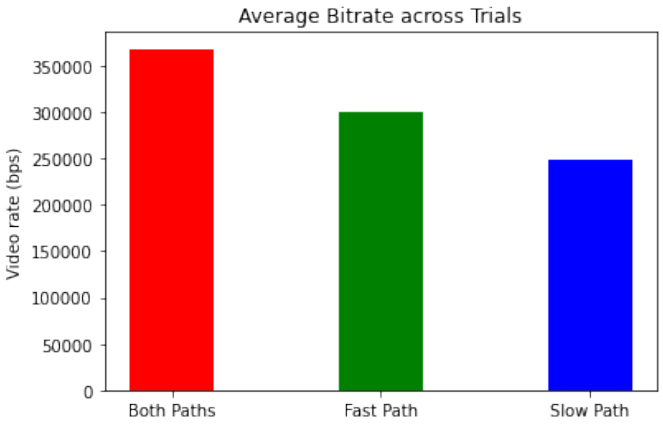

Fig. 4. For each trial, the average bitrate of the video playback was recorded over the entire 30-segment (approximately 120 seconds) period. At least 3 were run for each of the configurations, and the average of the average bitrate for all trials was plotted.

that the performance of the "Both Paths" configuration has a significantly better performance overall.

To ensure the results are statistically significant, we perform the Right-Tailed Two Sample Unpaired T-Test. When comparing the "Both Paths" and "Slow Path" samples, we obtain a p-value of 0.000264. When comparing the "Both Paths" and "Fast Path" samples, we obtain a p-value of 0.000688. In both cases, we find our results to be statistically significant at the 99% confidence level. Note that 3-4 samples were collected for each configuration, which is not the largest sample size; future work could be done to increase the amount of data collected.

```
2022/12/07 06:17:28.720 DEBUG   tls.handshake   choosing certificate    {"identifier": "10.0.2.2", "num_choices": 1}
2022/12/07 06:17:28.720 DEBUG   tls.handshake   default certificate selection results   {"identifier": "10.0.2.2", "subjects": ["10.0.2.2"], "mana
ged": true, "issuer_key": "local", "hash": "5016c30b5229465abbbd297f48331a47d15cf8d63f31fca307b68a86cd602074"}
2022/12/07 06:17:28.720 DEBUG   tls.handshake   matched certificate in cache    {"remote_ip": "10.0.1.1", "remote_port": "50582", "subjects": ["10
.0.2.2"], "managed": true, "expiration": "2022/12/07 17:30:27.000", "hash": "5016c30b5229465abbbd297f48331a47d15cf8d63f31fca307b68a86cd602074"}
2022/12/07 06:17:28.726 DEBUG   http.stdlib     http: TLS handshake error from 10.0.1.1:50582: local error: tls: bad record MAC
2022/12/07 06:17:28.736 DEBUG   events  event   {"name": "tls_get_certificate", "id": "0e9a2fed-41e2-41a3-8022-5c89bc102b6c", "origin": "tls", "da
ta": {"client_hello":{"CipherSuites":[4866,4867,4865,49196,49200,49195,49199,52393,52392,163,159,162,158,52394,49327,49325,49188,49192,49162,49172
,49315,49311,107,106,57,56,49326,49324,49187,49191,49161,49171,49314,49310,103,64,51,50,49245,49249,49244,49248,49267,49271,49266,49270,49239,4923
5,49238,49234,196,195,190,189,136,135,69,68,157,156,49313,49309,49312,49308,61,60,53,47,49233,49232,192,186,132,65,255],"ServerName":"","Supported
Curves":[29,23,30,25,24],"SupportedPoints":"AAEC","SignatureSchemes":[1027,1283,1539,2055,2056,2057,2058,2059,2052,2053,2054,1025,1281,1537,771,76
9,770,1026,1282,1538],"SupportedProtos":null,"SupportedVersions":[772,771],"Conn":{}}}}
```


Fig. 5. The output from *Caddy* shows that, while the QUIC requests from the client were arriving at the server, something is wrong with one of the MAC addresses (see the error "bad record MAC"), causing the TLS handshake to fail. Unfortunately, due to time constraints, I was unable to resolve this issue, which prevented me from collecting results to compare QUIC with HTTP/2 video streaming.

E. QUIC Video Streaming Results

As I mentioned in section III-D, my experimentation with QUIC began with finding a suitable server application that could run using the QUIC protocol, and my initial choice was Google's *quic_server* [10]. This ended up being quite an involved process. Since this server is intended primarily for integration testing, rather than as a production server, Google does not provide a standalone binary or package to install the *quic_server* from—instead, you must clone the entirety of the Chromium source and build the binary yourself. For context, the installation process requires over 100 GB of free disk space, and took well over two hours using the default CPU and RAM allocations provided by FABRIC nodes.

While the installation process was relatively smooth (aside from how long it took), upon finally installing *quic_server*, I began to see its limitations. While I was able to request a simple example HTML page as specified in their instructions, when I tried to host segments of the "Big Buck Bunny" dataset [13], the server was unable to start, citing that the files were missing HTTP headers. Upon further research, I found that this server implementation requires all statically hosted files to have the HTTP headers baked in; it does not insert the headers as the files are served. For a handful of HTML, CSS, and JS files, this would be easy enough, but since I am hosting thousands of 4-second video segments, manually labeling each file with headers was simply unfeasible. I considered writing a program to do this for me, but seeing as I could not find any other academic literature mentioning this process (only posts on the Google forums seem to acknowledge this limitation), I couldn't help but feel I was "barking up the wrong tree", so to speak.

Instead, I decided to switch to *Caddy* [11], a server application that claims to be the first to use QUIC by default. Using it solved many of the issues I faced with *quic_server*. First, it offers a binary that can quickly and easily be installed, instead of building from source. Since it uses QUIC by default, it also automatically installs the certificates needed for TLS to function (which *quic_server* requires you to do manually). However, my time with *Caddy* was not seamless. Upon requesting segments using the aforementioned fork of AStream [4], [5], *Caddy* output the error "bad record MAC" on the TLS handshake, and the logs about what exactly was wrong with the MAC address were limited, as shown in Figure 5. Clearly, since I had issues getting ARP and the MAC addresses to work in the previous phases of the project, something is wrong with my "workaround" solution. I made an attempt to fix the MAC addresses by adding additional "source" MAC address fields into the match/action tables of my P4 program, but this did not resolve the issue. Unfortunately, due to time constraints, I was unable to find a solution that allowed me to stream video over QUIC, and as such, this project was not successful in comparing the performance of QUIC vs HTTP/2 video streaming in the context of my P4 QoS program.

V. Summary, Conclusions, and Future Work

In the past, vertically-integrated router and switch designs have hidden their inner workings, and provided very limited configuration, often limiting the development of new protocols and the forms of traffic engineering that we can accomplish. With the advent of P4, we now have a mechanism for creating arbitrary programs that can run on these switches, which is entirely independent of the ossified protocols of the current internet, allowing us to route different application layer protocols (namely, HTTP/2 video streaming traffic) across links of, presumably, differing speeds, prioritizing these highly performance sensitive applications, over less performance sensitive ones.

Employing the use of the FABRIC testbed, a simple network topology was deployed, implementing this concept. My results show that when a single route shares its bandwidth between a video stream flow, and another flow of arbitrary cross-traffic to the same destination, as would be typical of traditional routing protocols, the performance of the video stream is significantly hampered compared to the custom P4 program, which is able to separate the flows across two distinct routes. This configuration improves the performance of the video stream by allowing both links to be fully utilized by the routers, despite the packets sharing the same destination IP address.

A. Limitations and Future Work

Of course, this work is not without its limitations. As was covered extensively in section IV-C, the custom P4 program used for testing currently does not implement the self-learning nature of ARP, requiring MAC address to be statically configured in each node's ARP cache. Given the minimal performance overhead required by ARP, this functionality was

omitted from the prototype, but future work could easily extend the program with such functionality.

Furthermore, as was noted in section III-C, our prototype is simplified from that of Bhat *et al.* [3], which uses P4 switches in conjunction with OpenFlow SDN switches to implement a more general design. Given that a large portion of this project was devoted as a learning exercise for exploring how to use the P4 language, we omitted the OpenFlow portion of this design, replacing them with P4 switches instead; future work may revolve around reintroducing OpenFlow back into the network topology while retaining the measured performance improvements.

Due to the time constraints imposed by this project, I was ultimately unsuccessful in testing the video streaming performance of QUIC versus HTTP/2 video streaming traffic. Future work would revolve around finding a solution to the TLS issues I faced—given the error messages I received, it is likely that a P4 program which fully implements a self-learning ARP algorithm instead would resolve this issue as well. From there, it would be trivial to use the artifacts I have created, alongside the logs from AStream, to compare the performance of QUIC and HTTP/2 video streaming in the context of providing QoS.

REFERENCES

[1] Cisco Systems Incorporated, "Vni complete forecast highlights," https://www.cisco.com/c/dam/m/en_us/solutions/service-provider/vni-forecast-highlights/pdf/Global_Device_Growth_Traffic_Profiles.pdf, accessed: 2022-09-27.

[2] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends," *IEEE Access*, vol. 9, pp. 87 094–87 155, 2021.

[3] D. Bhat, J. Anderson, P. Ruth, M. Zink, and K. Keahey, "Application-based qoe support with p4 and openflow," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2019, pp. 817–823.

[4] S. Arisu and A. C. Begen, "Quickly starting media streams using QUIC," in *Proceedings of the 23rd Packet Video Workshop*, ser. PV '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–6. [Online]. Available: https://doi.org/10.1145/3210424.3210426

[5] "quic-streaming," https://github.com/sevketarisu/quic-streaming, 2019, accessed: 2022-12-4.

[6] J. Crichigno, "Cybertraining: Virtual platform deployed for cybertraining purposes," http://ce.sc.edu/cyberinfra/cybertraining.html, accessed: 2022-11-13.

[7] "Behavioral Model (bmv2)," https://github.com/p4lang/behavioral-model, 2022, accessed: 2022-11-13.

[8] Apache Software Foundation, "Apache HTTP Server," https://httpd.apache.org/docs/2.4/, 2022, accessed: 2022-11-13.

[9] "Astream: A rate adaptation model for DASH," https://github.com/pari685/AStream, 2018, accessed: 2022-11-13.

[10] "Playing with QUIC," https://www.chromium.org/quic/playing-with-quic/, 2022, accessed: 2022-12-4.

[11] "Caddy," https://caddyserver.com/docs/, 2022, accessed: 2022-12-4.

[12] "P4C," https://github.com/p4lang/p4c, 2022, accessed: 2022-11-13.

[13] C. M. Stefan Lederer and C. Timmerer, "Dynamic adaptive streaming over http dataset," in *In Proceedings of the ACM Multimedia Systems Conference 2012*, Chapel Hill, North Carolina. February 22-24, 2012.

APPENDIX

All source code, P4 programs, *simple_switch_CLI* commands, and raw data used in this project are hosted on the following GitHub repository:

https://github.com/MCBulger2/CSE-534-p4-video-streaming-qos

To ensure the reproducibility of my experimental results on the FABRIC testbed, I have developed a Python Jupyter notebook. This notebook uses the FABLIB API to create a slice with the topology shown in Figure 2. It contains code for setting up the Apache web server, the AStream DASH video client, as well as the ability to generate the commands used by the control plane (*simple_switch_CLI*) to fill the routing tables of the switches.

Unless otherwise noted (such as the use of Mininet for the exploratory phase in section IV-A), all experimentation was done on the FABRIC testbed.