Evaluation of Packet Schedulers for Multipath QUIC
Evaluation of Packet Schedulers for Multipath QUIC

Alexander Rabitsch

© 2018 The author(s) and Karlstad University
This report is submitted in partial fulfillment of the requirements for the Master’s degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

__________________________
Alexander Rabitsch

Approved, June 19th 2018

__________________________
Advisor: Per Hurtig

__________________________
Examiner: Kerstin Andersson
Abstract

The Web has outgrown the transport mechanisms that have been used since its inception. Due to the increasing complexity of web pages in terms of both total size and number of individual resources, HTTP over TCP can no longer provide a satisfactory user performance. In recent years, much progress has been made in this area by evolving the web’s underlying mechanisms. Multipath QUIC (MPQUIC) is one such approach. MPQUIC is a new transport protocol which enables multihomed devices, such as smartphones, to aggregate their network interfaces in order to achieve greater performance. Additionally, MPQUIC is capable of multiplexing several data streams concurrently over a single connection, which can also provide performance benefits. This work began with a validation of our MPQUIC setup, which was performed by comparing MPQUIC to another multipath solution in a large set of experiments. The results show that MPQUIC is generally beneficial for the transfer time of large files, which corresponds with results from previous works. We additionally investigated ways to exploit MPQUIC’s multipath and stream features to achieve lower latencies for web pages via the means of packet scheduling. We implemented the Earliest Completion First (ECF) scheduler, and investigated how it compares against MPQUIC’s default path scheduler. The results indicate that the ECF scheduler is significantly more capable of handling heterogeneous network scenarios than the default scheduler, and can achieve higher throughput and lower latencies. Next, a Stream Priority scheduler was designed and implemented, which utilizes stream priorities to achieve lower completion times for select streams. The results from the investigation indicate that proper stream scheduling can significantly reduce download times of the prioritized resources. This effect was especially noticeable as path characteristics diverge. We also show that proper configuration of stream priorities is critical for such a scheduler, as a sub-optimal configuration yielded poor performance.

Keywords: Internet, Web, Transport, Transport protocol, Multipath, Multipath communication, Scheduling
Acknowledgements

First and foremost, I would like to thank my supervisor, Per Hurtig, for his invaluable support, ideas and feedback over the course of this work. Likewise, I would like to give special thanks to Anna Brunström and Karl-Johan Grinnemo for their guidance, and Jonas Karlsson for providing and maintaining the experimental testbed used for the evaluations.
Contents

1 Introduction 1
   1.1 Contribution ................................. 3
   1.2 Disposition ................................ 4

2 Background 5
   2.1 Computer Networking ............................. 5
   2.2 The Web .................................... 8
   2.3 Hypertext Transfer Protocol ... 9
      2.3.1 HTTP Multiplexing ..................... 11
   2.4 Transmission Control Protocol ............. 14
      2.4.1 TCP Handshake ........................... 15
      2.4.2 Byte Sequence Numbering .............. 16
      2.4.3 Congestion and Flow Control ........... 17
      2.4.4 Closing a Connection ................... 19
   2.5 QUIC ....................................... 19
      2.5.1 Transport Layer Security ............... 20
      2.5.2 QUIC Handshake .......................... 22
      2.5.3 Connection Migration .................... 23
      2.5.4 Stream Multiplexing ..................... 24
      2.5.5 Reliability ............................... 24
      2.5.6 Congestion Control ..................... 25
      2.5.7 Flow Control .............................. 27
      2.5.8 Closing a Connection .................... 27
   2.6 Multipath Transport: Concepts and Challenges 28
   2.7 Multipath TCP ................................. 30
      2.7.1 Path Management ......................... 31
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.1 Results</td>
<td>75</td>
</tr>
<tr>
<td>6.3 Conclusion</td>
<td>79</td>
</tr>
<tr>
<td>7 A Stream Priority Scheduler for MPQUIC</td>
<td>80</td>
</tr>
<tr>
<td>7.1 Design</td>
<td>81</td>
</tr>
<tr>
<td>7.2 Implementation</td>
<td>81</td>
</tr>
<tr>
<td>8 Evaluation of the Stream Priority Scheduler</td>
<td>84</td>
</tr>
<tr>
<td>8.1 Replaying Web Traffic</td>
<td>85</td>
</tr>
<tr>
<td>8.1.1 Results</td>
<td>86</td>
</tr>
<tr>
<td>8.2 Independent Web Requests</td>
<td>87</td>
</tr>
<tr>
<td>8.2.1 Results</td>
<td>87</td>
</tr>
<tr>
<td>8.3 Conclusion</td>
<td>95</td>
</tr>
<tr>
<td>9 Discussion</td>
<td>96</td>
</tr>
<tr>
<td>9.1 Path Scheduling</td>
<td>96</td>
</tr>
<tr>
<td>9.2 Stream Prioritization</td>
<td>97</td>
</tr>
<tr>
<td>10 Conclusion and Future Work</td>
<td>99</td>
</tr>
<tr>
<td>10.1 Future Work</td>
<td>99</td>
</tr>
<tr>
<td>References</td>
<td>101</td>
</tr>
<tr>
<td>A ECF Source Code</td>
<td>105</td>
</tr>
<tr>
<td>B Stream Priority Scheduler Source Code</td>
<td>110</td>
</tr>
<tr>
<td>B.1 Stream Selector</td>
<td>110</td>
</tr>
<tr>
<td>B.2 Stream Frame Bundling</td>
<td>111</td>
</tr>
<tr>
<td>C Control Script</td>
<td>114</td>
</tr>
</tbody>
</table>
Listings

2.1 HTTP request message. .................................................. 10
A.1 Source code for the ECF path scheduler. ........................... 105
B.1 Source code for the Stream Priority stream scheduler. ....... 110
B.2 Source code for the modified frame bundling process. ....... 111
C.1 Source code for the control script. ................................. 114
D.1 Source code for the replay program. ............................... 125

List of Algorithms

5.1 ECF path scheduler [31] ................................................. 61
5.2 ECF - Selecting the fastest and second fastest paths ......... 62
7.1 Stream Priority Scheduler .............................................. 82
List of Figures

2.1 Head-of-line blocking. ........................................ 13
2.2 Concurrent HTTP flows in HTTP/1.x and HTTP/2. .......... 13
2.3 UDP header format. ............................................ 15
2.4 TCP header format. ............................................ 15
2.5 A simplified illustration of a TLS handshake. ............... 21
2.6 QUIC 1-RTT and 0-RTT handshakes. ......................... 22
2.7 CUBIC window increase function. ........................ 26
2.8 Single-path and Multipath topologies. ....................... 29
2.9 TCP and MPTCP protocol stacks. ........................... 31
4.1 Overview of the virtualized network used in all experiments presented in this thesis. ...................................... 49
4.2 Download time ratio of the single-path and multipath protocols in lossless low-BDP scenarios. ............................ 55
4.3 Download time ratio of the single-path and multipath protocols in lossy low-BDP scenarios. ............................. 55
4.4 Download time ratio of the single-path and multipath protocols in lossless high-BDP scenarios. .......................... 55
4.5 Download time ratio of the single-path and multipath protocols in lossy high-BDP scenarios. .......................... 55
4.6 EAB of MPTCP and MPQUIC for lossless low-BDP scenarios. ........................................................... 56
4.7 EAB of MPTCP and MPQUIC for lossy low-BDP scenarios. ........................................................... 56
4.8 EAB of MPTCP and MPQUIC for lossless high-BDP scenarios. ......................................................... 56
4.9 EAB of MPTCP and MPQUIC for lossy high-BDP scenarios. ......................................................... 56
6.1 Mean download times against 95% confidence intervals in symmetric bandwidth scenarios. ..................................... 71
6.2 Mean download times against 95% confidence intervals in asymmetric bandwidth scenarios. ........................................ 72
6.3 Mean download times against 95% confidence intervals in highly asymmetric bandwidth scenarios. ........................ 73
6.4 Mean PLT against 95% confidence intervals in symmetric bandwidth scenarios. .................................................. 76
6.5 Mean PLT against 95% confidence intervals in asymmetric bandwidth scenarios. ............................................... 77
6.6 Mean PLT against 95% confidence intervals in highly asymmetric bandwidth scenarios. ...................................... 78
8.1 Mean render times against 95% confidence intervals in symmetric bandwidth scenarios. .................................... 88
8.2 Render times for all web pages in symmetric bandwidth scenarios. ................................................................. 88
8.3 Per-resource download times for the prioritized resources for all five web pages in symmetric bandwidth scenarios. ................................................................................................................ 88
8.4 Mean render times against 95% confidence intervals in asymmetric bandwidth scenarios. .................................. 89
8.5 Render times for all web pages in asymmetric bandwidth scenarios. ................................................................. 89
8.6 Per-resource download times for the prioritized resources for all five web pages in asymmetric bandwidth scenarios. ................................................................................................................ 89
8.7 Mean render times against 95% confidence intervals in highly asymmetric bandwidth scenarios. ........................................ 90
8.8 Render times for all web pages in highly asymmetric bandwidth scenarios. ............................................................. 90
8.9 Per-resource download times for the prioritized resources for all five web pages in highly asymmetric bandwidth scenarios. ................................................................................................................ 90
8.10 Mean completion times for the critical rendering resources against 95% confidence intervals in symmetric bandwidth scenarios. ................................................................. 92
8.11 Completion times of critical rendering resources for all web pages in symmetric bandwidth scenarios. ................................................................. 92
8.12 Per-resource download times for the prioritized resources for all five web pages in symmetric bandwidth scenarios. .......................... 92
8.13 Mean completion times for the critical rendering resources against 95% confidence intervals in asymmetric bandwidth scenarios. ................. 93
8.14 Completion times of critical rendering resources for all web pages in asymmetric bandwidth scenarios. ................................. 93
8.15 Per-resource download times for the prioritized resources for all five web pages in asymmetric bandwidth scenarios. ....................... 93
8.16 Mean completion times for the critical rendering resources against 95% confidence intervals in highly asymmetric bandwidth scenarios. .... 94
8.17 Completion times of critical rendering resources for all web pages in highly asymmetric bandwidth scenarios. .............................. 94
8.18 Per-resource download times for the prioritized resources for all five web pages in highly asymmetric bandwidth scenarios. .................... 94
## List of Tables

2.1 Network stack layers and their responsibilities. ........................................ 6  
2.2 UDP and TCP features. ................................................................. 15  
4.1 Link parameter space for the validation. ....................................................... 50  
6.1 Link parameter space for the evaluations. ....................................................... 68  
6.2 Percentage of packets sent on WLAN for symmetric scenarios in terms of bandwidth. ................................................................. 71  
6.3 Percentage of packets sent on WLAN for asymmetric scenarios in terms of bandwidth. ................................................................. 72  
6.4 Percentage of packets sent on WLAN for highly asymmetric scenarios in terms of bandwidth. ................................................................. 73  
6.5 Web pages used during the evaluation. ......................................................... 75  
6.6 Percentage of packets sent on WLAN for symmetric scenarios in terms of bandwidth. ................................................................. 76  
6.7 Percentage of packets sent on WLAN for asymmetric scenarios in terms of bandwidth. ................................................................. 77  
6.8 Percentage of packets sent on WLAN for highly asymmetric scenarios in terms of bandwidth. ................................................................. 78  
8.1 Stream priorities. ................................................................. 85
1 Introduction

The Web has evolved significantly from its inception in the late 1980's. Over the years, web pages have increased in size and complexity; a modern web page may contain hundreds of individual pieces which must be downloaded before a web browser can fully load it. At the same time, the mechanisms behind the web has not evolved at the same speed as the web itself. Transport protocols which have traditionally been used for the web are becoming less capable of providing a satisfying user experience. The increasing complexity of the average modern web page, coupled with the inefficiencies of the web’s underlying transport protocols, may lead to long load times. Simultaneously, the load time of a web page is crucial for the user experience, and may consequently negatively impact the revenue it may generate.

According to studies by Google which investigated the impact of the performance of mobile web pages, users have a low tolerance for slow load times [3]. They found that the average time to completely load a mobile web page is 15 seconds. For 70% of web pages, it took more than five seconds before the web browser began rendering, and seven seconds to load all elements visible before scrolling. Meanwhile, they found that over half of all users will abandon a page if it takes longer than 3 seconds to load. Their studies also show that probability of abandonment only increases along with the load times. As the load time increased from 1 second to 10 seconds, the probability of abandonment increased by 123%. Mobile sites which load in 5 seconds increased the viewability of advertisements by 25%, increased the length of the average session by 70% and earned up to 2 times more from advertisements compared to sites which load in 19 seconds [22]. A similar study by Akamai, where user data was collected from visits to online retailers, corroborate these results [1]. They found that a mere 100 ms delay increase in load time can reduce conversion rates by up to 7%. Delays of one and two seconds reduced conversion rates by up to 21.8% and 36.5%, respectively, and also increased the bounce rate by up to 103%. They also found that a delay of two seconds can decrease the average session length by up to 51%. Bottom
line - web page performance is crucial for the user experience, which in turn may directly affect the revenue of a business. There is, as such, much to be gained from optimizing web load times.

Web page performance can be improved in many ways. The simplest solution (which is, however, still not trivial), is to optimize the web page itself to ensure that a web browser may request additional resources in the most efficient way. This is not a general approach; optimizations must be done on an individual basis. It may also not be enough to reduce page load times to a satisfactory level. A more general approach, which is consequently much more complex, is an evolution of the communication protocols used by the web. Google has been very active in this area. Their first notable attempt, SPDY [4], is a now deprecated experimental networking protocol which aimed to achieve lower web page load times by modifying the behavior of the traditional web protocol, HTTP [24]. Ideas from the SPDY protocol were eventually incorporated into a new version of HTTP, namely HTTP/2 [5]. QUIC [28] is another attempt to achieve greater web performance, yet again spearheaded by Google. QUIC is an experimental protocol closely tied to HTTP/2, which aims to eventually replace TCP [37] as the de facto transport protocol for the web, and is currently undergoing standardization. Both HTTP/2 and QUIC use a stream concept, which allows for multiplexing of several independent data streams concurrently over a single connection. The stream concept specifically addresses inefficiencies between the interaction of HTTP and TCP. HTTP over TCP is incapable of efficient multiplexing of data streams due to the head-of-line blocking phenomenon caused by packet reordering at the receiver. The stream concept used by HTTP/2 over QUIC not only avoids head-of-line blocking between independent data streams, but also gives rise to ample opportunities to further reduce web page load times, by for example prioritizing the sending of resources required by a web browser to begin rendering over other resources.

Another method to reduce web page load times is multipath communication, which allows a multihomed device (a device with multiple networking interfaces) to aggregate its
network interfaces to achieve greater performance. Going forward, such devices are expected to become increasingly common. For example, smartphones and tablet computers can typically interface with both Wi-Fi and Mobile Broadband access points. By aggregating the different network paths provided by the two access points, the load time of web pages can be significantly reduced. However the opposite can also be true; the page load time may actually increase when using multiple network interfaces compared to only using the faster interface. This is because the characteristics of the two paths can differ significantly, which may result in packet reordering at the receiver, leading to head-of-line blocking. Considering mobile devices such as smartphones, such scenarios are not unlikely. For example, a smartphone user may move to a location with poor Wi-Fi reception, meaning that the use of the Wi-Fi interface may no longer be beneficial. Thus, a pivotal challenge for multipath transport protocols is the scheduling of data on the different paths. Ideally, a multipath transport protocol should be able to fully aggregate the capacity of all available paths, but only send on paths on the condition that it is beneficial for the total transfer. This thesis examines whether web performance can be improved using advanced scheduling techniques for Multipath QUIC (MPQUIC) [8].

1.1 Contribution

The goal of this work has been to investigate ways to reduce the latency of web transfers using both the stream concept of HTTP/2 and QUIC, and QUIC’s recent experimental multipath extension, known as MPQUIC. MPQUIC is as of yet in an early stage, and as such, the opportunities for improvements are great. Due to its recency, this work begins with an experimental validation of MPQUIC by replicating the experiments from the initial design and evaluation paper [9]. Since the way data is scheduled is of utmost importance for the performance of MPQUIC, in terms of both throughput and latency, we also investigate the possibility of improving the packet scheduler. The possibilities of both path scheduling (which path should be used for transmission) and stream scheduling (which order should
data be sent) for MPQUIC are explored. In short, the contributions of this work are:

- A validation of the MPQUIC prototype.
- Implementation and evaluation of the Earliest Completion First (ECF) [31], a state-of-the-art path scheduler, for MPQUIC.
- An investigation on the possibility and implications of a Stream Priority scheduler for MPQUIC.
- A discussion on how to move forward with the knowledge gained from the previous steps.

1.2 Disposition

The structure of the remainder of this document is as follows. We begin with Section 2, which provides the necessary background information required to grasp the topics covered in this thesis. Section 2 includes an overview of the web and its underlying transport protocols, such as HTTP, TCP and QUIC, and introduces the concept of multipath transport along with some relevant realizations, such as MPTCP and MPQUIC. Section 3 discusses related previous work. Section 4 covers the experimental validation of MPQUIC by the replication of previous experiments [9]. In Section 5 the ideas behind the design and implementation of the ECF path scheduler for MPQUIC are discussed. The evaluation of the ECF scheduler is then covered in Section 6. In Section 7, the design ideas behind a Stream Priority scheduler and its implementation are provided. Section 8 covers the experimental evaluation of the Stream Priority scheduler. Next, Section 9 discusses the methods used in this work, and the yielded results. We conclude with Section 10. Additional material is appended in Appendices A-D.
2 Background

In order to understand the contribution of this work, we must first have an understanding of the web, the performance issues that it may be susceptible to, and thus how the performance of the web can be improved. First, the core concepts behind the web need to be understood. Thus, we begin with a brief overview of the Internet architecture and the web. Next, we describe some of the main mechanisms behind the web. To give an idea of known performance issues of the web, and possible solutions to these issues, we introduce relevant application layer and transport layer protocols, such as HTTP, TCP and QUIC. Multipath is another method which may be used to improve the performance of web applications. However, utilization of multiple paths concurrently introduces a new set of challenges to overcome, in particular how data can be scheduled over available paths efficiently. Thus, we next introduce the core ideas behind multipath communication. Finally, to give an idea of how multipath communication is realized in practice, a selection of multipath transport protocols are covered, including Multipath TCP (MPTCP) [20] and MPQUIC.

2.1 Computer Networking

A computer network consists of a set of nodes which are connected via links. In order to communicate, nodes are able to transmit messages over the links. The Internet is a packet switched network. Unlike a circuit switched network (such as a telephone network), nodes are not allocated a dedicated channel for the communication. Instead, messages are divided into packets, which need to be forwarded between nodes until they may be received at the receiving node. Packets consist of headers, which are used by nodes in order to route the packets to the receiver, and the payload, which contains the actual message. Typically, the process required for computers to communicate over the Internet tend be described as an abstraction consisting of various layers, each with a different responsibility. This abstraction is generally known as the TCP/IP protocol stack. In this section an introduction to the
TCP/IP protocol stack is given. Table 2.1 provides an additional overview of the different layers.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Name</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>Physical layer</td>
<td>Pushes bits into the link</td>
</tr>
<tr>
<td>L2</td>
<td>Link layer</td>
<td>Handles communication between adjacent nodes</td>
</tr>
<tr>
<td>L3</td>
<td>Network layer</td>
<td>Routes packets over the network</td>
</tr>
<tr>
<td>L4</td>
<td>Transport layer</td>
<td>Handles process to process communication</td>
</tr>
<tr>
<td>L5</td>
<td>Application layer</td>
<td>Provides a common language for the end-points</td>
</tr>
</tbody>
</table>

Imagine the following scenario:

1. Alice wants to send a letter to Bob. In order for Bob to understand what is written in the letter, Alice needs to compose the letter in a language that Bob can understand (L5).

2. Bob might live together with other people in his home. To ensure that the letter is delivered to the correct person, Alice writes Bob’s name on the letter (L4).

3. Alice might live in a different city from Bob, and can not deliver the letter to Bob’s home directly; she must rely on a postal service to deliver the letter. Thus, Alice writes Bob’s address on the letter to ensure that it can be delivered to Bob’s home. The two cities are not located next to each other, so the letter can not be transported directly between the cities. Thus, the letter may be transported to various distribution centrals in between the two cities before it is delivered to Bob’s address. For the sake of efficiency, the letter should be transported on the shortest path (L3).

4. When transporting the letter between stops, a postal service worker does not necessarily need to be concerned about how to get to Bob’s address; the worker only needs to know how to deliver the letter to the next stop (L2).

5. Finally, Alice must put the letter in a letter box so that the postal service may pick it up (L1).

Parallels can be drawn between the scenario in the previous paragraph and the TCP/IP...
stack. When two processes exchange data, there needs to be a way for the processes to understand each other. For example, when browsing the Internet, your web browser needs to send something in order for the server to understand which web page you want to visit. The server also needs to respond with something else which can be understood by your web browser. This is an example of the use case for the Hypertext Transfer Protocol (HTTP) [5, 24], which is an application layer protocol. Protocols in the application layer provide the processes with a common language with which they can communicate. The responsibilities of the application layer corresponds to (1) in the scenario described above.

The application layer protocols are built on top of the transport layer. Computer systems typically run several different processes concurrently. When two systems are communicating, it is thus important that the messages arrive to the correct process. This is the responsibility of the transport layer, which corresponds to (2). Each process running on a system is given its unique port number, which can be used by remote systems to address the process. Additionally, a computer network is not reliable; packets may become corrupted or lost. In the scenario with Alice and Bob, this corresponds to the possibility that Alice’s letter might get lost at a distribution central or become damaged during the transport. The transport layer can provide functionality in order to handle packet corruption and losses. Traditionally, there has been two de facto transport layer protocols. The first is the User Datagram Protocol (UDP) [35]. UDP is a bare-bones protocol which only provides error detection (in case of data corruption) and a field for port numbers. Due to its simplicity, UDP is well suited for applications that require low latencies and can tolerate packet loss or reordering, and can be used as a substrate for other transport protocols. The Transmission Control Protocol (TCP) [37] is a more advanced protocol which offers additional features, such as reliability against packet losses. TCP is described in greater detail in Section 2.4.

The network layer is responsible for guiding packets through the network in a process called routing. This corresponds to (3) in the example with Alice and Bob. Nodes whose main purpose is to route packets are called routers. Routing is done using the Internet
Protocol (IP) [36]. IP addresses are used to identify the nodes in the network. Routers maintain a routing table which is used to identify the best path for a packet to take. Based on the destination IP address of the packet and the routing table, a router determines an appropriate output link. Note that a router may have algorithms in place which changes the routing table over time. This means that packets may take different paths even though the sender and receiver do not change. This can lead to packets being received in a different order than they were sent. Also note that routers do not have infinite memory. When a router receives a packet, it must choose an appropriate output link to forward the packet. If a router receives more packets than it can output (the output link is a bottleneck), the memory allocated to buffer packets can become full. This is typically handled by dropping any further incoming packets. This phenomenon is called network congestion. Additionally, interference on the links between the nodes on the network may result in bit-flips (data corruption).

The link layer is responsible for the communication over the links between adjacent nodes in the network and corresponds to (4) in the example with the letter. The communication over a link may be handled differently depending on the properties of the link, for example if the link is wired or wireless. The link layer includes protocols such as Ethernet and IEEE 802.11 (Wi-Fi). Finally, there is the physical layer, which is responsible for the transmission of bits on a link. This can be seen as the equivalent to (5) in the example with Alice and Bob. For the sake of brevity, the link and physical layers will not be covered.

2.2 The Web

The World Wide Web (the Web) is a network of web resources, such as documents and images, which can be accessed over the Internet. Each resource in the web is identified via a unique Uniform Resource Identifier (URI) [6]. A Uniform Resource Locator (URL, or web address) is a form of URI which not only identifies a resource, but also provides the means to locate it in the network. The URI http://www.example.com/index.html is an example of
a URL, which includes a scheme, a hostname and the name of the resource file. In this case, the scheme (http://) indicates that the HTTP protocol is used, and the hostname (www.example.com) is an alternate human readable name for the host’s IP address.

Before processing a URL, a web browser must first resolve the hostname in the URL into a proper IP address. This can be done using the Domain Name System (DNS) [32]. DNS is a hierarchical system of domains and sub-domains. The hierarchy of domains can be read in an ascending order from left to right in a hostname. At the top of the hierarchy are the top-level domains, such as .com or .net. Thus, the host name www.example.com is in the domain www, which is a sub-domain of example, which in turn is a sub-domain of the top-level domain com.

Each domain is represented by at least one name server [32], which the web browser may query in order to resolve a name to a proper IP address. For example, www.example.com resolves into the IP address 93.184.216.34, which is the IP address of the server hosting the resource requested in the URL. Once the hostname has been resolved, the browser may attempt to access the resource. Most commonly, this is done by sending a request to the host of the resource via HTTP over TCP.

2.3 Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is an application layer protocol which has become the backbone for communication over the web. HTTP allows a client, such as a web browser, and a server to communicate by exchanging HTTP messages. These messages consist of request messages from the client, and response messages from the server. A client may interact with a server via different request-methods. Request-methods are accompanied by a Request URI, which is used to identify the resource which the request should apply to. Below is a list of available HTTP methods [24]:

• **CONNECT**: Used to establish a tunnel to the server (for example SSL or TLS).
• **DELETE**: Request that the server deletes the resource identified by the specified URI.
• **GET**: Request the retrieval of the resource identified by the specified URI, such as a web page.
• **HEAD**: Works identically to the GET method, however, the response does not contain a message-body.
• **OPTIONS**: Request information about available options for the resource at the specified URI.
• **POST**: Used to send data to the server. For example, used to post data on a bulletin board.
• **PUT**: Store an entity at the resource identified by specified URI. If there already exists a resource at the specified URI, it will be replaced.
• **TRACE**: Invokes a remote loop-back test.

Each HTTP request includes a request-line. The request-line consists of a request-method, a Request-URI, as well as the protocol version. A request also includes a request-header, which can be used in order to provide additional information about the request which is not covered by the request-method.

### Listing 2.1: HTTP request message.

```
GET http://example.com/ HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:58.0) Gecko/20100101 Firefox/58.0
Accept: text/html, application/xhtml+xml, application/xml; q=0.9, */*; q=0.8
Accept-Language: en-US, en; q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```
Imagine that a user wants to visit the web site http://example.com/ using the Firefox web browser on their Ubuntu machine. After resolving the hostname in the URL, the web browser constructs a HTTP request using the GET method. The request is then sent to the server indicated by the resolved IP address. The message seen in Listing 2.1 is an example of what such a request could look like. The request contains a request line indicating a GET method, the URI (http://example.com/), the HTTP version (HTTP/1.1) as well as a request-header. The request-header contains information about the web browser (Firefox running on Ubuntu), options such as accepted file formats, language and encoding and other parameters for the connection. After receiving the request message from a client, the server responds with a response message. The first line of the response message is the status-line, which consists of the protocol version and the status-code. Status-codes consist of three digits, where the first digit describes the category of the response. Status-lines also include a reason-phrase, which is a textual description of the status-code. The categories of responses are [24]:

- 1xx: Informational - Signaling that the request has been received and will continue to be processed.
- 2xx: Success - Signals that the request was successful.
- 3xx: Redirection - Signals that the client must take additional actions in order to complete the request.
- 4xx: Client Error - Signals that the request was invalid.
- 5xx: Server Error - Signals that the server failed to fulfill a valid request.

Response messages also contain a response-header, which allows the server to include additional information about the response which is not covered in the status-line.

2.3.1 HTTP Multiplexing

Today, web pages have a tendency to contain a large amount of elements, all of which need to be individually requested via HTTP. In order to reduce the download times of such web
pages, several elements can be requested concurrently.

Earlier versions of HTTP (HTTP/1.0 and HTTP/1.1) are severely limited in regards to support for concurrent requests. HTTP/1.0 only allows a single request per TCP connection [24]. In order to concurrently request resources from a server, it is thus needed to establish a new TCP connection per element when using HTTP/1.0. HTTP/1.1 introduced request pipelining, which allows clients to make multiple requests on a single TCP connection without having to wait for a response [5, 24]. However, HTTP/1.1 pipelining is susceptible to head-of-line blocking. HTTP/1.1 relies on in-order delivery of responses. If a packet carrying one HTTP flow is lost, or if packets are received out of order, all following HTTP flows will be held up waiting for the flow which is first in line. Refer to Figure 2.1 for an illustration of the head-of-line blocking phenomenon. The head-of-line blocking issue has caused HTTP/1.1’s pipelining feature to become largely ignored. In order to achieve concurrency, clients still tend to open up several TCP connections at once when using HTTP/1.1. While TCP also relies on in-order delivery, by opening up a new connection per request, it is ensured that a lost packet over one connection does not impact any other HTTP flow. Note that TCP also allows for persistent connections, allowing a TCP connection to be reused for multiple requests.

The problem with opening up several TCP connections is that most elements on a web page are small. TCP requires end-points to establish a connection before HTTP messages can be sent. This means that a relatively large period of the lifetime of an HTTP flow is spent establishing a connection instead of transmitting data. Furthermore, TCP is designed to ramp up the rate which data is sent over time. Since most HTTP flows are short, the rate does not have much time to increase, which results in poor utilization of available bandwidth. For longer flows, another effect is observed. TCP is not tailored for multiple connections over the same path. This results in unfairness against other TCP connections, and may lead to congestion and packet losses. In essence, HTTP interacts poorly with TCP, which in turn has resulted in poor performance for modern web sites.
Figure 2.1: Head-of-line blocking. The packet which is first in line is lost. All other packets must wait until the retransmitted packet is received before they can be processed.

Figure 2.2: Concurrent HTTP flows in HTTP/1.x and HTTP/2. HTTP/1.x uses multiple TCP connections, whereas HTTP/2 uses HTTP multiplexing over a single TCP connection.

HTTP/2 (previously also known as HTTP/2.0) aims to address these issues by supporting true multiplexing of HTTP flows over a single connection. Figure 2.2 illustrates the different concurrency schemes of HTTP/1.x and HTTP/2. Multiplexing is achieved in HTTP/2 by associating each HTTP flow with a distinct stream. A single HTTP/2 connection may carry multiple streams concurrently. HTTP/2 also allows streams to be established and closed by either end-point. A server may ‘push’ responses it anticipates the client will need before actually receiving a request in order to reduce latency.

Streams are used to carry various HTTP/2 frames, which contain HTTP requests/responses as well as various control messages. Each stream is associated with a Stream ID. Streams opened by the client use an odd-numbered Stream ID, whereas streams opened by the server will be given an even-numbered Stream ID. The end-points may negotiate the maximum number of streams that may be concurrently active using SETTINGS frames. A client may assign the priority of a stream, which is used by the server as input when deciding which stream should be transmitted next. Streams are generally considered independent from each other. However, a stream may be given a stream dependency in order to express that the stream is dependent on another stream. Note that the priorities do not ensure
that a collection of streams are transmitted in a particular order, and does not apply if there are enough resources to support all streams concurrently.

As previously mentioned, TCP requires in-order delivery of packets before application data can be properly processed. Since HTTP/2 carries several HTTP streams over a single TCP connection, HTTP/2 can still experience head-of-line blocking from the transport-layer. HTTP/2 uses a flow control mechanism which aims to limit the interference between streams. The flow control mechanism operates on a per-stream basis and for the connection as a whole. A sender maintains a flow control window for each stream as well as for the entire connection. The flow control window determines how much data can be sent per stream/connection. By transmitting a frame, the sender must reduce the size of the send window according to the size of the frame. The sender may not transmit a frame if it means that the flow control window of the stream/connection will be exceeded. The receiver may increase the flow control window of the sender by transmitting a \texttt{WINDOW\_UPDATE} frame, in which the receiver advertises how much additional data it is prepared to receive. The HTTP/2 standard [5] does not specify which events should trigger the transmission of a \texttt{WINDOW\_UPDATE}, as HTTP/2 intends to allow pluggable flow control algorithms.

### 2.4 Transmission Control Protocol

The Transmission Control Protocol (TCP) [37, 12] is the \textit{de facto} transport layer protocol of the Internet. As shown in Table 2.2 and in Figures 2.3 and 2.4, TCP is a much more complex protocol than the bare-bones alternative, UDP. Compared to UDP, TCP offers a wide set of features, which is used to compensate for weaknesses of the underlying layers. TCP requires the end-points to set up a connection using a 'handshake', which is covered in Section 2.4.1. TCP uses \textit{sequence numbers} in order to provide reliability against packet losses and corruption (see Section 2.4.2). Additionally, to not overwhelm the network or the receiver, TCP can limit the rate at which data is transmitted using congestion and flow control. These mechanisms are covered in Section 2.4.3. Finally, the connection teardown
is covered in Section 2.4.4.

### Table 2.2: UDP and TCP features.

<table>
<thead>
<tr>
<th>Feature</th>
<th>UDP</th>
<th>TCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection establishment</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Reliable data transfer</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Error detection</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Flow control</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Congestion control</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

#### 2.4.1 TCP Handshake

Before application data can be transmitted over TCP, a connection between the end-points needs to be established. This is because various features of TCP, such as the reliability and congestion control mechanisms require that information about each data stream is
maintained by both end-points. A TCP connection may be reused for different data streams. In order to avoid confusion, the packets used to establish a connection will use random initial sequence numbers. A connection is established via TCP’s three-way-handshake. During the handshake, the end-points exchange initial sequence numbers. A client triggers the handshake by sending a packet with the SYN control bit set in its header, indicating that the end-points should synchronize their initial sequence numbers. Such packets are called SYN packets. Upon receiving the SYN packet, the server responds with a packet where the SYN and ACK control bits are set (SYN-ACK). Finally, upon receiving the SYN-ACK, the client responds with a packet where the ACK control bit is set (ACK). Application data can not be sent until the TCP handshake has been completed.

2.4.2 Byte Sequence Numbering

Each byte sent over a TCP connection is identified by a sequence number. Each packet carries a sequence number field in its header which is used to identify which bytes in the data sequence are carried by the packet. The sequence number indicates the amount of payload bytes sent so far, and does not include headers. Both end-points maintain their own set of sequence numbers.

When an end-point receives a packet, the receiver must acknowledge that the packet has been received and properly processed. This is done by sending a packet with the acknowledgement number in the header set to the next expected byte. Such packets also set the ACK control bit. Packets with this flag set are generally referred to as ACKs. Since an ACK carries the sequence number of the next expected byte which has not yet been received, it cumulatively acknowledges that all bytes with lower sequence numbers have been received properly. The use of sequence numbers allows a sender to know about packet losses; if the sender does not receive an ACK for a packet that it has sent within a reasonable time, it is deemed lost. This time is based on the round-trip-time (RTT), and triggers a retransmission timeout (RTO). Lost packets may then be retransmitted. Retransmitted
packets carry the same bytes as the originals, and thus keep the same sequence numbers. This scheme also allows the receiver to know if packets have been received out of order. Other methods, such as SACKs or delayed ACKs, will not be covered here for the sake of brevity.

### 2.4.3 Congestion and Flow Control

Network congestion occurs when packets arrive at a bottleneck link quicker than the link can handle. Thus, TCP implements a congestion control mechanism. The idea is to maximize the utilization of available bandwidth while avoiding congestion. TCP’s standard congestion control [7] consists of four different algorithms. TCP’s congestion control is based on a *Congestion Window* (typically abbreviated as *cwnd*), which determines how much data a sender can send per RTT. In conjunction with the congestion control, TCP also features a flow control mechanism which is used to ensure that a sender does not send more data than the receiver can handle. The flow control is based on a *Receive Window* (*rwnd*), which also puts a limit on how much data a sender can transmit. The congestion window is maintained by the sender, while the receiver maintains the receive window, which is advertised to the sender. TCP uses whichever is lowest of the two to determine how much data can be sent. Additionally, a slow start threshold (*ssthresh*) state variable is used in order to determine which algorithm is used for the congestion control:

- If *cwnd* is less than *ssthresh*, the congestion control uses the *Slow Start* algorithm.
- If *cwnd* is greater than *ssthresh*, the *Congestion Avoidance* algorithm is used.
- If *cwnd* is equal to *ssthresh*, either slow start or congestion avoidance can be used.

At the beginning of a connection, the *ssthresh* is typically set to an arbitrarily high number, and is reduced as a response to congestion. The exact algorithms used by TCP may vary as the congestion control mechanism uses a modular design. The following describe the algorithms from the current standard [7].

The Slow Start algorithm is used at the beginning of a TCP connection. At this point,
TCP does not know the characteristics of the path between the end-points. Thus, in order to avoid congestion, the slow start phase starts with a congestion window size set to a conservative value. For each ACK received during the slow start phase, the congestion window is increased by one packet. Since the receiver responds with an ACK for each packet sent, the congestion window is essentially doubled per RTT. This allows TCP to quickly ramp up the sending rate to an appropriate level. The slow start phase lasts until either a loss is detected, indicating congestion, or ssthresh is reached. When the congestion window exceeds ssthresh, the congestion control switches to the Congestion Avoidance algorithm. During this phase, the congestion window is increased by one packet per RTT. This phase lasts until congestion is detected. During the Slow Start and Congestion Avoidance phases, ssthresh is set to max(FlightSize/2, 2) upon detecting packet loss due to RTO. FlightSize is the number of outstanding packets in the network that has yet to be acknowledged. The congestion window is also reduced to 1 packet as a reaction to congestion.

The Fast Retransmit algorithm attempts to predict packet losses before an RTO occurs. When receiving packets out-of-order, either due to re-ordering or packet loss, the receiver responds with duplicate ACKs. Out-of-order packets will have sequence numbers higher than that which the receiver expects. When replying to such packets, the receiver reuses the acknowledgement number from the previously sent ACK. This is because the next expected byte has not yet been received. Such ACKs are thus known as duplicate ACKs. Upon receiving the missing packet, the receiver may immediately ACK all received bytes until the next expected byte in sequence. Incoming duplicate ACKs are used by the fast retransmit algorithm. After receiving three duplicate ACKs, the sender assumes that the packet has been lost and retransmits the packet immediately. At this point, ssthresh is set to max(FlightSize/2, 2), and cwnd = ssthresh + 3. The congestion control also switches to the Fast Recovery algorithm. For each additional duplicated ACK received,

\footnote{Note: in actuality, the congestion control is based on the \textit{maximum segment size} (bytes) of the connection. In this text, 'packet' is used for the sake of simplicity.}
the congestion window is incremented by one. Once the next ACK which acknowledges previously unacknowledged data has been received, the congestion window will be set to ssthresh, and the congestion avoidance algorithm can take over.

2.4.4 Closing a Connection

Once the end-points have no more data to send the connection can be closed. This can be done in two ways. Normally a connection is concluded using a 'FIN handshake', which allows the connection to be terminated gracefully. A graceful termination means that any pending data is delivered before the connection is terminated. An immediate termination can also be forced by transmitting packets with the RST control bit set. RST is typically used as a last resort when an end-point has lost the state of a connection, for example after a server crash.

A 'FIN handshake' is initiated by sending a packet with the FIN control bit set. The receiving end-point must ACK the received FIN packet and respond with a FIN packet of its own. The initiator must then ACK the receiving end-point’s FIN. A connection termination may be initiated by either end-point.

When initiating a 'FIN handshake', it is not guaranteed that the other end-point is done transmitting. Thus, the initiator must be prepared to handle incoming packets even after it has sent a FIN. Only once both end-points have acknowledged the FIN of the other end-point can the connection be considered terminated.

2.5 QUIC

QUIC (formerly also known as Quick UDP Internet Connections) [28] is an experimental transport protocol suggested by Google. QUIC is specifically designed to improve the performance of the web. This is done by offering features such as faster connection establishment (compared to TCP) and innate multiplexing of independent data streams to avoid head-of-line blocking. Furthermore, QUIC also offers built-in security. QUIC is tunneled
over UDP for the sake of deployability; it allows QUIC packets to pass through middleboxes without the risk of being blocked. Additionally, changes to the operating systems running on the end-points are not required to run the protocol [28]. QUIC is currently undergoing standardization by the Internet Engineering Task Force (IETF) [26]. This section is structured as follows. First, QUIC’s built-in security and low-latency handshake are covered (Sections 2.5.1 and 2.5.2). Next, Section 2.5.3 describes QUIC’s connection migration feature. Stream multiplexing over QUIC is covered in Section 2.5.4. QUIC’s reliability and congestion control schemes are introduced in Sections 2.5.5 respectively 2.5.6. Section 2.5.7 details QUIC’s flow control mechanism. The section concludes with how a QUIC connection is terminated (Section 2.5.8). Note that QUIC is currently undergoing standardization and is as such rapidly evolving. This means that there may be some discrepancies between QUIC as it is described here and the most current versions.

2.5.1 Transport Layer Security

Transport Layer Security (TLS) is a cryptographic application layer protocol, which provides a connection with a secure channel. In this context, a secure channel refers to a channel which provides features such as authentication, integrity and confidentiality. TLS is often used to secure an HTTP connection. This scheme is typically called Hypertext Transfer Protocol Secure (HTTPS) or simply HTTP over TLS. QUIC uses an integrated TLS-like solution, not only for providing security, but also to exploit TLS’s cryptographic handshake in order to provide low-latency connection establishment. This section primarily focuses on TLS version 1.3 [39]. Any discrepancies with other variations are not considered.

In order to establish a secure channel, the end-points of a connection are required to perform a cryptographic handshake, in which the end-points agree to which parameters shall be used. Most importantly, the handshake provides the end-points with the cryptographic keys used to secure the channel. A TLS connection can perform key exchanges using different key exchange schemes [39]:

20
- Diffie-Hellman (DHE)
- Elliptic Curve Diffie-Hellman (ECDHE)
- Pre-shared key (PSK)
- PSK with DHE
- PSK with ECDHE

Figure 2.5: A simplified illustration of a TLS handshake. A client and server negotiate connection parameters using the ClientHello and ServerHello messages. Next, the client confirms that the cryptographic handshake has been completed successfully using a MAC. Finally, the two end-points may begin transmit application data.

As shown in Figure 2.5, a TLS handshake begins with a ClientHello message. In the ClientHello message, the client specifies the protocol parameters it supports, and includes cryptographic information about the key exchange. The server responds with a ServerHello message, which establishes the parameters for the connection, including cryptographic information which can be used by both end-points to derive the keys. Once the parameters for the connection have been established, the server typically transmits a certificate along with a signed key, which are used in order to authenticate the server. Optionally, the server may request authentication from the client, which is performed similarly to how the server was authenticated. Finally, the client responds with a Message Authentication Code (MAC), which is used to confirm that the handshake has been completed successfully.

A PSK derived from one connection can be reused for future connections. This allows
TLS 1.3 offer a 0-RTT handshake variant, meaning that a client can transmit application data along with the ClientHello message. During the first flight of early data, the PSK is used for authentication (rather than a certificate) and to encrypt the application data. The rest of the handshake proceeds as normally. Since the 0-RTT handshake relies on non-ephemeral keys, forward secrecy for early data can not be provided.

### 2.5.2 QUIC Handshake

QUIC handshakes range from 1-RTT for first time connection establishment, down to 0-RTT for connections to known peers. QUIC uses the TLS 1.3 [44] handshake as a basis for its low-latency connection establishment. In order to minimize the latency for establishing a QUIC connection, the transport and TLS handshakes are combined into one, which is performed using **STREAM** frames over a dedicated stream. QUIC provides reliability and ordered delivery for the TLS handshake messages. In return, the TLS handshake can be used to carry the desired QUIC transport parameters [28].

![QUIC Handshake Diagram](image)

Figure 2.6: QUIC 1-RTT and 0-RTT handshakes. The 0-RTT handshake uses cached information from a previous connection which allows the client to transmit application data, such as an HTTP request, immediately without having to wait for a response.

A client must wait until the handshake is completed before beginning to transmit data. If a client connects to a server with which the client has not previously established a
connection, the handshake can be completed within 1-RTT. By caching information from this connection, such as a PSK, any subsequent handshakes can be reduced to 0-RTT [44]. 0-RTT handshakes allows the client to send application data immediately after the handshake packet [28]. TLS also offers a handshake variation which initiates a new key exchange. This variation can be used by QUIC in order to validate that a client truly owns the address it claims [44].

2.5.3 Connection Migration

QUIC identifies connections using a 64-bit Connection ID, which is included in the QUIC header [28]. The Connection ID allows the end-points to migrate from one network path to another without having to close the connection, in case the quality of the current path degrades. An end-point which receives packets from a new source address, but with a familiar Connection ID, can immediately start sending packets via the new path [28]. However, end-points must verify that packets can be received on the new path before fully committing to the switch [28]. This means that an end-point must limit the rate at which it sends data until the new source address has been validated. QUIC validates a path by sending a \texttt{PING} frame containing a validation token to the peer via the new remote address. Upon receiving a \texttt{PING} frame, the peer must then respond with a \texttt{PONG} frame, which contains the same token. If the validation is unsuccessful, the end-point terminates the connection. In the case of a successful validation, the end-point may begin to increase the rate at which it sends data. The end-point may also refresh any validation tokens it may have issued its peer [28]. Empty \texttt{PING} frames (which do not contain a validation token) may additionally be used periodically by an end-point in order to check the reachability of its peer. Empty \texttt{PING} frames are simply acknowledged as normal instead of requiring the transmission of a corresponding \texttt{PONG} frame.

Since connection migration is susceptible to reordering due to varying path conditions, a host must be prepared to receive packets from several source addresses at the same time.
Thus, the path used to transmit data is based on which path has received the highest packet number [28].

2.5.4 Stream Multiplexing

QUIC is capable of multiplexing several data streams over a single connection. In contrast to TCP, QUIC is designed in such a way that a lost packet only blocks the specific streams that it carries; other streams may carry on unaffected [28]. Thus, QUIC can provide HTTP/2 multiplexing void of head-of-line blocking.

QUIC achieves multiplexing by using \texttt{STREAM} frames to carry application data. The \texttt{STREAM} frame contains a Stream ID field which is used to identify which stream the frame belongs to. A single QUIC packet may carry several different frames, including \texttt{STREAM} frames from different streams [28]. In order to maximize efficiency, a QUIC packet should carry as many frames as possible. Note that all streams carried by a packet will be blocked by a loss. QUIC does not deal with losses on a packet-level, meaning that lost packets are not retransmitted. Instead, lost frames may be retransmitted by being re-bundled into new outgoing packets [28]. This implies that frames are independent from the packets which carry them.

Stream prioritization can have a large impact on application performance. QUIC relies on the application to provide a preferred order in which stream frames are sent. Stream 0, which is used to establish the connection, is always prioritized over other streams until a connection has been properly established [28].

2.5.5 Reliability

QUIC uses ACK based loss detection and congestion control similar to that of TCP, with a few key differences [27]. Similarly to TCP, all packets carry a sequence number in the header. QUIC uses ACK frames, which are sent by receiver in order to confirm that the data has been received and properly processed. If the receiving end-point takes too long
to respond to a packet with an ACK frame, an RTO event occurs. A single ACK frame may cumulatively acknowledge one or more ranges of packets [28]. Each ACK frame carries a *Largest Acknowledged* field, which represents the largest packet number received so far, regardless of gaps. QUIC ACK frames also carry an *ACK Block Section*, containing up to 256 *ACK Blocks*, which represent the various packet ranges that are acknowledged.

As covered in Section 2.5.4, QUIC does not retransmit lost packets. Upon a detected loss, the frames carried by the lost packet are instead re-bundled into another packet with a new packet sequence number. Thus, packet sequence numbers do not repeat over a connection. This also means that the sequence number carried by a packet is not indicative of the order of the data carried in the packet. In order to determine the order of the data in a stream, the *STREAM* frames contain *Offset* and *Length* fields [28].

### 2.5.6 Congestion Control

QUIC features pluggable congestion control mechanisms. The *quic-go* [15] implementation of QUIC uses the CUBIC [25, 40] congestion control scheme by default, which is a variant of TCP’s congestion control designed for high-speed networks. CUBIC is also used as the default congestion control mechanism in the TCP implementation in Linux. CUBIC aims to remain fair to standard TCP connections while modifying the congestion window growth function to be more scalable than TCP’s standard congestion control. See Section 2.4.3 for an overview of TCP’s standard congestion control algorithms.

As the name suggests, CUBIC uses a cubic function when increasing the congestion window in the congestion avoidance phase, and does not rely on RTT like the standard algorithm does. The window size is increased according to 

\[ w_{\text{cubic}}(t) = C(t - K)^3 + w_{\text{max}}, \]

where \( C \) is a constant scaling factor, \( t \) is the time since the beginning of the congestion avoidance phase and \( w_{\text{max}} \) is the maximum window size before it was last reduced [25]. \( K \) is calculated as 

\[ K = \frac{3}{\beta} \sqrt{w_{\text{max}}^2 \beta / C}, \]

where \( \beta \) is CUBIC’s multiplication decrease factor used for window reduction [25].

25
CUBIC runs in three different modes depending on the current size of the congestion window (cwnd) [40]:

1. The **TCP-friendly Region**.
2. The **Concave Region**.
3. The **Convex Region**.

The TCP-friendly ‘region’ is not actually a region represented in CUBIC’s cubic function, but is a secondary window increase function which is used in order to provide throughput at least as high as standard TCP in scenarios where standard TCP typically performs well [40]. CUBIC attempts to estimate the window size of standard TCP according to $w_{tcp}(t) = w_{max} \times \beta + 3 \times \frac{1-\beta}{1+\beta} \times \frac{t}{rtt}$. CUBIC is in the TCP-friendly region when this secondary window increase function yields a larger window than the cubic function. The congestion window is as such set according to $cwnd = \max(w_{cubic}, w_{tcp})$.

![Figure 2.7: CUBIC window increase function, with the origin at $w_{max}$. The X-axis depicts the window increment based on the current window size, represented by the Y-axis. When $cwnd < w_{max}$ and CUBIC is not in the TCP-friendly region, CUBIC is in the concave region (left side). When $cwnd > w_{max}$ and CUBIC is not in the TCP-friendly region, CUBIC is in the convex region (right side). When CUBIC is in the TCP-friendly region, a different window increase function is used, which is not depicted here.](image)

As shown in Figure 2.7, CUBIC is in the concave region when CUBIC is not in the TCP-friendly region and cwnd is less than $w_{max}$, and in the convex region when CUBIC is not in the TCP-friendly region and cwnd is greater than $w_{max}$ [40]. During both the concave
and convex region modes, cwnd is increased according to $cwnd = cwnd + \frac{w_{cubic}(t+rtt)-cwnd}{cwnd}$ for each received ACK. Due to the inclusion of $w_{max}$ when calculating $w_{cubic}(t)$, the cwnd increments behave differently in the different modes. The concave region mode initially increases cwnd fairly aggressively, but slows down as it gets closer to $w_{max}$. The opposite is true for the convex region mode; CUBIC slowly probes for additional bandwidth initially and accelerates the larger cwnd grows [25].

At the event of an RTO, CUBIC follows the standard TCP congestion control, except it sets $ssthresh = cwnd \times \beta$. During the first congestion avoidance phase after an RTO, CUBIC calculates $w_{cubic}(t)$ where $K = 0$, $w_{max}$ is set to the initial value of cwnd at the beginning of the current congestion avoidance phase and $t$ is the time that has passed since the start of the congestion avoidance phase.

When CUBIC detects congestion using other methods the following values are set:

1. $w_{max} = cwnd$
2. $ssthresh = max(cwnd \times \beta, 2)$
3. $cwnd = cwnd \times \beta$

### 2.5.7 Flow Control

QUIC uses a stream based flow control mechanism similar to that of HTTP/2 (see Section 2.3.1). A receiver must continuously advertise how much data it is willing to accept per stream as well as for the entire connection. The advertisement of receive windows is done using MAX_STREAM_DATA or MAX_DATA frames. MAX_DATA dictates how much data can be sent on the connection as a whole, whereas MAX_STREAM_DATA advertises how much data the receiver is willing to accept on a specific stream.

### 2.5.8 Closing a Connection

A QUIC connection can be terminated in three different ways: due to a timeout, immediate close or a stateless reset [28].
During the QUIC handshake, the end-points negotiate an idle timeout value. If no data has been transmitted or received for longer than the idle timeout value, the connection is closed.

An immediate close may be used for a graceful termination of the connection and is initiated by sending a closing frame. After sending a closing frame, the sender enters the closing state. During this state, the sender of the closing frame responds to all incoming packets with another closing frame. Upon receiving a closing frame, an end-point may choose to respond with a closing frame of its own, however this step is optional.

An end-point enters the draining state upon receiving a closing frame or if an idle timeout occurs. During the draining state, the end-point may not transmit any packets. End-points in the closing state may transition into the draining state upon receiving confirmation that its peer is in the draining state. Both the closing state and draining states last for three times the RTO value.

The stateless reset can be used in order to terminate a connection immediately. The stateless reset is used similarly to TCP’s RST (see Section 2.4.4). To perform a stateless reset, an end-point transmits a packet containing a stateless reset token derived from the QUIC handshake.

2.6 Multipath Transport: Concepts and Challenges

In more recent years, as multihomed devices such as smartphones have become more and more common, the concept of aggregating multiple available network paths has gained traction. A device is considered multihomed when it owns two or more separate IP addresses. Typically, two communicating processes use a single set of IP addresses, and thus, the packets are imagined routed over a single path. The idea of multipath is to be able to use more than one IP address per communication, and thus be able to use multiple paths. Multipath is based on the principle of resource pooling. Resource pooling is the principle of making a collection of resources behave as if they were a single resource [47]. In the
context of a network, multiple network links can be aggregated together to achieve higher efficiency in regards to throughput and latency, and to provide additional reliability and flexibility. For example, the web browser of a smartphone can aggregate the capacity of the WLAN and Mobile Broadband interfaces in order to reduce the time required to download a web page. Other use-cases include signaling traffic, which may benefit from the extra robustness against network failures that multiple paths offer. Figure 2.8 illustrates the difference between a single-path connection (A) and multipath connection (B). If the path between the end-points fails in topology A, so does a potential connection between the end-points. If one of the paths in topology B fails, the communication may still be carried over the remaining path. Furthermore, the end-points may aggregate the bandwidth of both paths in topology B in order to achieve greater performance than would be possible using only a single path.

Figure 2.8: Single-path and Multipath topologies. Topology A provides end-points with a single-path (IP 1 - IP 2). In contrast, the end-points in topology B can aggregate the paths (IP 1 - IP 3) and (IP 2 - IP 4) for increased robustness and performance.

The multipath transport layer protocol Stream Control Transmission Protocol (SCTP) [42] is already commonly used for signaling traffic within telephone networks, where multiple paths are used as a failover measure. QUIC [28] brings a similar set of features to end devices. Meanwhile, extensions to TCP, SCTP and most recently, QUIC, have been developed in order to take advantage of Concurrent Multipath Transfer (CMT) [2, 8, 20].

Multipath communication introduces a new set of challenges which needs to be dealt with. A common problem is that the utilization of multiple paths can cause unfairness to single-path transport protocols such as TCP. Even though the end-points may send or receive data through completely separate links, there is no way to guarantee that the
different paths do not share a bottleneck link somewhere in the network. This means that a multipath connection can take up an unfair share of the available network resources compared to single-path connections (ideally, all connections should have an equal share of resources). This problem requires special consideration when designing congestion control mechanisms for multipath. The characteristics of the different paths utilized by a single connection also pose a challenge; paths may be widely asymmetrical in terms of bandwidth, latency and congestion. Such differences must also be considered when designing congestion control. More importantly, an end-point must carefully consider which packet should be sent over which path, lest it result in severe performance issues due to sending packets on a slower path. Thus, multipath protocols include a scheduling component, which determines which path is used for transmission. It bears to be mentioned that concurrent multipath communication is not always beneficial, depending on the circumstances. Multipath is generally not considered beneficial for short lived transfers, or when the different paths are highly asymmetrical. For example, the download time of a web page can actually increase rather than decrease by utilizing two highly asymmetric paths in terms of bandwidth and delay. In such a scenario, the faster of the two paths may be able to complete the entire transmission in shorter time than it takes to transmit a single packet on the slower path. As such, the scheduling component is crucial to the performance of a multipath protocol. Ideally, a scheduler should be designed to only aggregate multiple paths when it is beneficial, and remain single-path when it is not. Turn to Section 3 for details on existing packet scheduling approaches.

2.7 Multipath TCP

Multipath TCP (MPTCP) [20] is a multipath extension for TCP. MPTCP provides concurrent multipath transfer functionality by allowing TCP to open so called subflows. A subflow is roughly equivalent to a normal TCP connection, however, many subflows may be aggregated over a single MPTCP connection. Figure 2.9 illustrates a comparison between
TCP and MPTCP. MPTCP operates using 'faux-headers' implemented using TCP options.

An MPTCP connection is initiated on a single subflow using a three-way-handshake similar to TCP. However, the packets used for the handshake additionally carry the MP\_CAPABLE option. This option is used to verify that both end-points support MPTCP. The MP\_CAPABLE option is also used to exchange tokens which can be used to authenticate additional subflows. Section 2.7.1 covers how MPTCP establishes new paths. Reliability, congestion control and flow control are introduced in Sections 2.7.2, 2.7.3 and 2.7.4 respectively. Next, the schedulers used in the Linux kernel implementation of MPTCP are introduced in Section 2.7.5. Finally, Section 2.7.6 covers how an MPTCP connection is terminated.

![Figure 2.9: TCP (left) and MPTCP (right) protocol stacks. MPTCP aggregates multiple TCP subflows over different paths.](image)

### 2.7.1 Path Management

After an MPTCP connection has been established the end-points may open up new subflows. The simplest way for a host to communicate that it owns an additional address is to open a subflow directly over that path. A host may also learn about the additional addresses of its peer (if there are any) through address signaling. A host may advertise additional addresses by transmitting packets with the ADD\_ADDR option over an already established subflow. Previously advertised addresses can be revoked using the REMOVE\_ADDR option.
A subflow is opened by transmitting a SYN packet which includes the MP_JOIN option. The MP_JOIN option contains the token stemming from the handshake over the initial subflow. In order to prevent replay attacks, the token is hashed and is transmitted along a nonce - an arbitrary number which is only used once.

The ADD_ADDR, REMOVE_ADDR and MP_JOIN options carry an Address ID. The Address ID is used to identify the source address of the packet. This is useful in case a middlebox such as a NAT changes the original source address in the IP header. The Address ID may be thus used for address removal without requiring knowledge of the true address. The Address ID is also useful to prevent end-points from setting up duplicate subflows.

2.7.2 Reliability

The simplest way to provide reliability for MPTCP would be to use the TCP sequence numbers in order to indicate the packet sequence regardless of the subflow which the packet was sent on. However, this approach is vulnerable as middleboxes may block traffic with gaps in the sequence space or otherwise tamper with the sequence numbers [38]. Thus, each subflow operates using its own sequence number space and each packet includes a data sequence mapping which maps the payload to the sequence of bytes for the entire MPTCP connection. Similarly, the ACKs on a subflow can only acknowledge packets on a specific subflow. In order to be able to cumulatively ACK packets on a connection-level, MPTCP uses Data Acknowledgements (Data ACK). The Data Sequence Signal (DSS) option is used to carry the data sequence mapping and Data ACKs.

2.7.3 Congestion Control

When utilizing multiple paths, care must be taken in order to achieve fairness at bottleneck links while still achieving resource pooling. As multiple paths may share a single bottleneck link, using a standard TCP congestion control for each subflow would not result in fairness to normal TCP flows. Thus, MPTCP uses coupled congestion control for all subflows. For
a description of the TCP congestion control mechanism, see Section 2.4.3.

The Opportunistic Linked-Increases Algorithm (OLIA) [29] is a multipath variant of the standard TCP congestion control mechanism which couples the increases of the congestion windows of each MPTCP subflow. OLIA modifies the behavior of the congestion avoidance phase, while otherwise behaving like the standard TCP congestion control. During the congestion avoidance phase, the window size of path \( r \) is increased according to Equation (2.1) per ACK received over the path, where \( w \) is the window size, \( rtt \) is the round-trip-time and \( \mathcal{R}_u \) is the set of available paths [29]. \( \alpha \) is calculated using Equation (2.2) where \( \mathcal{M} \) is the set of paths with the largest window sizes and \( \mathcal{B} \) is the set of presumed best available paths, at any point in time [29]. \( \mathcal{B} \setminus \mathcal{M} \) represents good paths that has not yet been fully utilized [29]. Through the \( \alpha \) term, OLIA can increase the window sizes of good paths that are not fully utilized by redirecting the traffic from fully utilized paths. After detecting a loss on a subflow, its window size is halved.

\[
\frac{w_r}{rtt_r^2} \left( \sum_{p \in \mathcal{R}_u} \frac{w_p}{rtt_p} \right)^2 + \alpha_r/w_r \quad (2.1)
\]

\[
\alpha_r = \begin{cases} 
\frac{1}{|\mathcal{R}_u|}, & \text{if } r \in \mathcal{B} \setminus \mathcal{M} \neq \emptyset \\
-\frac{1}{|\mathcal{R}_u|}, & \text{if } r \in \mathcal{M} \text{ and } \mathcal{B} \setminus \mathcal{M} \neq \emptyset \\
0, & \text{otherwise} 
\end{cases} \quad (2.2)
\]

### 2.7.4 Flow Control

Similarly to TCP, MPTCP uses a receive window to provide flow control. This receive window is shared between the subflows. By using a single receive window for the whole connection, subflows are not limited by its own receive window. This allows MPTCP to utilize its paths in whatever way it wishes as long as the receive window for the whole connection is not surpassed. While it would be possible to use unique receive windows for each subflow, there would be no actual benefit to do so, as it could unnecessarily throttle
the faster paths.

2.7.5 Packet Scheduling

When using multiple paths, a packet scheduler is required in order to determine which path a packet should be sent. The MPTCP implementation in the Linux kernel [33] provides a modular scheduler infrastructure which gives users a choice between various different packet schedulers. This section covers the standard schedulers that come with the Linux implementation.

The simplest scheduler is the Round-Robin (RR) scheduler. Using RR, subflows will take turns transmitting data in a cyclical fashion. RR is a very simple scheduler which does not take path characteristics into consideration, which can lead to poor performance unless the paths are symmetrical. If the sender is able to fill the congestion windows of all subflows, the scheduling becomes ack-clocked [34], meaning that new packets will be scheduled on whatever subflow happens to have space available as its congestion window empties.

The Lowest-RTT-First (LowRTT) is the default scheduler of MPTCP. The LowRTT scheduler prefers the path with the lowest estimated RTT. Once the congestion window for that path has been filled, the scheduler moves on to the path with the second lowest RTT, and so on. Similarly to the RR scheduler, LowRTT also becomes ack-clocked if the congestion windows of all subflows are filled [34].

Finally, the Linux implementation of MPTCP includes the redundant scheduler. The redundant scheduler duplicates the traffic on all available paths in order to achieve low latency at the cost of bandwidth.

2.7.6 Connection Teardown

In normal TCP, a connection is terminated using a 'FIN handshake' where both end-points must advertise that they have no more data to send (see Section 2.4.4). This implies that
the normal TCP ‘FIN handshake’ may only close the specific subflow used to send the FIN packets. This allows MPTCP to close subflows it no longer wants to use during a connection. If all subflows used in a connection have been closed this way, the MPTCP connection waits for a timeout until declaring the connection dead. In order to close all active subflows in an MPTCP connection simultaneously, MPTCP uses the DATA_FIN mechanism. An end-point may signal that it has no more data to send by setting the F flag in the DSS option. This initiates a ‘DATA_FIN handshake’ equivalent to a ‘FIN handshake’ for the entire connection. Once both end-points have Data ACKed each other’s DATA_FINs, the connection is considered terminated, after which each subflow is terminated using standard ‘FIN handshakes’.

2.8 Multipath QUIC

Multipath QUIC (MPQUIC) [9] is an extension of the QUIC transport protocol which enables QUIC connections to aggregate multiple network paths similarly to how MPTCP enables multipath for TCP. QUIC is already capable of utilizing multiple paths for a single connection via the connection migration mechanism, as described in Section 2.5.3. However, QUIC is not capable of effectively utilizing more than one path concurrently for a single connection. MPQUIC aims to go beyond QUIC’s connection migration mechanism in order to provide true CMT functionality [8].

MPQUIC natively supports fair congestion control mechanisms inspired by MPTCP. Additionally, MPQUIC is less susceptible to middlebox tampering than MPTCP, which results in fewer compromises and a simpler design. Since MPQUIC supports multiplexing of streams, it is more robust against head-of-line blocking and is capable of using stream-based packet scheduling techniques. Section 2.8.1 gives an introduction to how MPQUIC maintains multiple paths, whereas Sections 2.8.2 and 2.8.3 briefly covers MPQUIC’s default congestion control and packet scheduler.
2.8.1 Path Management

A path in MPQUIC is represented by a unique four-tuple consisting of [8]:

- IP source address
- IP destination address
- Source port
- Destination port

Similarly to QUIC, an MPQUIC connection is established over a dedicated stream on an initial path, where the TLS handshake is performed. After completing the handshake, MPQUIC avoids using the initial path for data transfer. Instead, MPQUIC may establish a ‘duplicate’ of the initial path using the same IP addresses but different port numbers. During the handshake, both end-points advertise how many paths may be used for the connection. The lowest advertised value will then dictate how many additional paths may be used for the connection [8]. A more detailed description of how the handshake is performed can be found in Section 2.5.2. MPQUIC can exploit QUIC’s 0-RTT handshake in order to establish low-latency connections on new paths. MPQUIC is capable of allowing both end-points to open new paths. However, in the current implementation of MPQUIC server-initiated paths are not used [9].

Each path is identified using a unique Path ID. Packets transmitted after a connection has been established include the Path ID in its header [8]. The four-tuple associated with a path may change over time. MPQUIC does this by utilizing QUIC’s connection migration mechanism. Unlike QUIC, which migrates the entire connection to another path, MPQUIC can use this mechanism to migrate a path to another path. In MPQUIC, this ability is thus called path migration [8]. In order to migrate a path, an end-point can simply send a packet using the same Path ID on a different path. The four-tuple maintained by the receiver will then be updated according to the most recent packet received.

MPQUIC also introduces new frame types used to manage multiple paths. The PATHS frame contains information about the active paths of the sender. This information includes
the Path ID, local address and the senders perceived RTT of the path. The PATHS frame can be used in order to detect potential connectivity issues and to give both end-points a global view of each path [8]. Additionally, an end-point may use the ADD_ADDRESS frame in order to advertise that a new address is available, and the REMOVE_ADDRESS to signal that a previously available address has become unavailable [8].

A core idea in QUIC is that frames should be independent from the packets which carry them. To remain compliant with this idea, MPQUIC also considers frames independent from the path that they are sent on [8]. This allows MPQUIC to retransmit frames over different paths. Each path maintains its own set of packet sequence numbers. The ACK frame has thus been modified in order to allow for acknowledgements on a per-path basis by including the corresponding Path ID [8]. The changes to the ACK frame implies that acknowledgements for packets on one path may be sent on a different path. While this opportunity provides additional scheduling flexibility, such a decision can also impact the perceived RTT of a path. The acknowledgement of frames are typically used for the estimation of RTT. However, since the reception of ACK frames are no longer guaranteed to experience a ‘true’ round-trip on a single path, acknowledgements may no longer be suitable for RTT estimation.

2.8.2 Congestion Control

Using a single-path congestion control scheme in a multipath setting can possibly result in unfairness against single-path protocols. Thus, the current MPQUIC implementation has opted for OLIA as the standard congestion control scheme [9]. Turn to Section 2.7.3 for a description of the OLIA congestion control scheme.

2.8.3 Packet Scheduling

In the current implementation of MPQUIC, the packet scheduler is based on the Lowest-RTT-first (also known as the default scheduler) used by the MPTCP implementation in
the Linux kernel [9]. For a description of the Lowest-RTT-First scheduler, see Section 2.7.5.

The MPQUIC scheduler has a few key differences to the default scheduler used in MPTCP. As mentioned in Section 2.8.1, MPQUIC is not limited to retransmitting frames on the same path. In contrast, MPTCP is required to retransmit packets on the same path in order to avoid being blocked by middleboxes [9]. Since there is no way of knowing the RTT of a path which has not yet transmitted data, the MPTCP scheduler will duplicate all traffic of another path until the RTT of the path becomes known [9].
3 Related Work

In our investigation of how web page performance can be improved via the means of packet scheduling, we have taken inspiration from scheduling techniques presented in previous works, which are presented here. This work distinguishes between two different types of packet scheduling: *path scheduling* and *stream scheduling*. A path scheduler is responsible for choosing which path a packet shall be sent on. Naïve schedulers such as Round-Robin (RR) and First-Come-First-Served (FCFS) do not take path characteristics into consideration, which may lead to significant performance issues when paths are asymmetrical. More complex schedulers thus typically prioritize sending on the ‘best’ path according to metrics such as path RTT. A stream scheduler may be used for transport protocols such as CMT-SCTP and MPQUIC, where it determines the order streams shall be sent, and which streams shall be sent over which path. In their non-CMT variants, streams in SCTP and QUIC are typically considered independent from the path which is used to carry them. This means that data from one stream may be bundled along with data from any other stream in a packet, which is then sent over whichever path the scheduler chooses. If the connection only has a single path this behaviour is reasonable, however, it may be sub-optimal when utilizing multiple paths concurrently. While streams are independent from each other, head-of-line blocking may still occur on a per-stream basis. By sending stream data over different paths in regards to latency, there is an increased chance that the stream experiences head-of-line blocking. In order to avoid inter-stream head-of-line blocking, it may thus be beneficial to carefully schedule stream data by sending it over an appropriate path. This section briefly discusses some previously proposed path and stream schedulers and their evaluations.

Paasch et al. [34] evaluated the Lowest-RTT-First scheduler for MPTCP against the RR scheduler. The comparison was performed using three variations of the Lowest-RTT-First scheduler: LowRTT (standard), LowRTT-RP (Retransmission and Penalization) and LowRTT-BM (Bufferbloat Mitigation). LowRTT-RP uses an *Opportunistic Retransmission*
and Penalization (ORP) algorithm. Opportunistic Retransmission allows a subflow blocked by the receive window to retransmit unacknowledged packets originally sent over a different path in order to quicker overcome head-of-line blocking [38]. Additionally, LowRTT-RP penalizes the blocked path in order to prevent it from exceeding the receive window again by decreasing its congestion window. ORP is currently used in the Linux kernel implementation of MPTCP. LowRTT-BM uses another algorithm which aims to mitigate the effect of bufferbloat by limiting the rate at which data is sent [38]. The measurements were carried out using both real-world and emulated environments, consisting of a scenario where a client and server exchange data over two MPTCP subflows. In the emulated environment, the characteristics of the two paths used to send data were varied by applying a space-filling design across two distinct classes: low-bandwidth-delay-product (low-BDP) and high-bandwidth-delay-product (high-BDP) settings. For the real-world environment, WLAN and 3G interfaces were used for the two paths. Paasch et al. investigated the performance of the schedulers for bulk transfers, which mainly benefit from high throughput, as well as for latency sensitive applications.

In the evaluation of bulk transfers, Paasch et al. investigated the effect path scheduling has on connections with bounded as well as unbounded receive buffers. In order to investigate the impact of schedulers on latency sensitive connections, Paasch et al. investigated the delay-jitter for blocks of data by sending at a constant rate over the two paths, and measuring the required time for each block to be fully received. The data rate was then varied. In the emulated environment, the impact of the schedulers were evaluated by calculating the aggregation benefit [10] of the measured goodputs. It was shown that the RR scheduler achieves an aggregation benefit on-par with the standard LowRTT variation for bulk traffic. Paasch et al. attribute this phenomenon to the ack-clocking effect mentioned in Section 2.7.5. In Low-BDP scenarios, their results showed little difference between the schedulers. Both LowRTT and RR performed worse than the other schedulers in high-BDP scenarios. According to Paasch et al., this is likely because LowRTT and RR cannot handle receive
window limitations as well as the other two schedulers. In the real-world environment, the goodput achieved by transfers of large files was measured with both bounded and unbounded (very large) receive buffers. Their findings show that the choice of scheduler is more important when the receive buffer is bounded compared to unbounded buffers. For unbounded receive buffers, all four schedulers achieved high goodputs. The differences between the schedulers are more prominent for bounded receive buffers. Their findings indicate that LowRTT-RP and LowRTT-BM variants perform slightly better than the other schedulers when the receive buffers are bounded. In unbounded scenarios, the sender is generally not limited by receive windows, meaning that the sender can continue to send data unhindered by re-ordering at the receiver. In bounded scenarios, the receive buffer of the receiver is more likely to fill up due to head-of-line blocking, resulting in more frequent receive window limitations at the sender, and LowRTT-RP and LowRTT-BM are better equipped to handle such limitations [34]. Their results for latency sensitive connections show a significant disparity between the RR scheduler and the LowRTT variants, where the former is outperformed in both the emulated and real world environments. According to their findings, the performance of RR is closer to the other schedulers as the data rate increases. Paasch et al. explains this result as RR sends data over the worst path more often than the LowRTT variants. At lower data rates, the Low-RTT schedulers can exclusively utilize the best path, whereas higher data rates ’force’ the LowRTT variants to transmit data over the worse path due to congestion window limitations [34]. Noticeably, their findings indicate that the LowRTT-BM scheduler achieves lower delays than the other schedulers for an unbounded data rate as it can better avoid bufferbloat in the network [34].

De Coninck et al. [9] showed that MPQUIC outperforms MPTCP for long transfers when using the LowRTT scheduler. Their evaluation was based on measurements from an emulated environment where a client and server exchange data over two separate paths. The bottleneck links of these paths were varied using an experimental design approach across
four classes: low-BDP-no-loss, low-BDP-losses, high-BDP-no-loss and high-BDP-losses. Both MPTCP and MPQUIC used the OLIA congestion control (see Section 2.7.3 for a description) and encryption. The MPQUIC connections were established using 0-RTT handshakes. De Coninck et al. investigated the performance of the two multipath protocols in terms of latency via short file transfers, and throughput via long file transfers. They then compared the measured download times. Additionally, the download times for the multipath protocols were also compared against their respective single-path variant in order to calculate the experimental aggregation benefit [9]. Their results show that MPQUIC outperforms MPTCP for all classes and download sizes using the LowRTT scheduler. However, the results also show that multipath transfer is generally not beneficial for short transfers. As mentioned in Section 2.6, when paths are asymmetrical, the faster path may be able to complete the entire transfer faster than it takes to send packets over the slower path. This phenomenon is accentuated as transfer sizes get smaller. Thus, it is more beneficial to stay single-path for short transfers. Another reason is because when the transfer sizes are small, the congestion window of the fastest path is rarely, if ever, full. Since the LowRTT scheduler only begins to transmit on another path once the congestion window of fastest path is full, MPTCP and MPQUIC will function virtually the same as their single-path variants (using the fastest path). Thus, we do not see an improvement in these scenarios.

Yang et al. [48] introduced the Out-of-order Transmission for In-order Arrival Scheduling (OTIAS) scheduler for MPTCP, which aims to provide shorter delays as compared to the LowRTT scheduler in asymmetric network scenarios. While the LowRTT scheduler prioritizes the fastest path, it unconditionally moves on to the second fastest path as soon as the fastest path is blocked by its congestion control. If the disparity between the RTTs are high enough, this behavior may be detrimental to the performance of MPTCP. Unlike the LowRTT scheduler, OTIAS allows a packet to be scheduled on a path even if its congestion window is full. Packets scheduled on a blocked path may then be transmitted
once the congestion window of the path recovers. Packets are scheduled on paths based on the shortest estimated transfer time, taking previous scheduled data, path capacity and RTT into consideration. The result is that OTIAS schedules packets in-order, but may transmit them out-of-order in terms of data sequencing. The idea is that the packets arrive in-order at the receiver due to the differences in path delay. Yang et al. evaluated OTIAS against the LowRTT scheduler in a physical testbed with two multihomed hosts. They used both bulk and delay sensitive traffic for their evaluation. Their results indicate that OTIAS can significantly reduce the receive buffer occupancy as opposed to LowRTT, and can thus achieve a higher throughput due to being blocked by the flow control less often.

Sarwar et al. [41] proposed a Delay Aware Packet Scheduling (DAPS) algorithm for CMT-SCTP aimed at reducing receive buffer blocking due to packet reordering when concurrently transmitting over asymmetric paths. DAPS uses an approach similar to that of OTIAS, however, unlike OTIAS, which schedules on a per-packet basis, DAPS creates a schedule for a batch of packets at a time, and then follows the schedule until the next batch. First, the goal is to find the ideal number of packets to be sent over all paths, taking the forward delays and capacity of the paths into consideration. Next, DAPS attempts to find the ideal order to send these packets over the paths to minimize reordering at the receiver. This is done by mapping the sequence numbers of packets to the different paths, taking the transfer time of each packet into consideration. Sarwar et al. evaluated DAPS against a RR scheduler using simulations where a sender and receiver exchange data over asymmetric paths. Their results show that DAPS achieves a lower receive buffer occupancy and higher goodput than the RR scheduler, even as the accuracy of the forward delay estimation used for the scheduling decisions deteriorates.

Kuhn et al. [30] suggested simplifications of the DAPS algorithm and further evaluated the performance of the scheduler over a more advanced set of simulation cases than [41], by
varying path capacity and RTT as well as the size of the receive buffer. Their results show that DAPS is better equipped to handle scenarios with small receive buffers than RR, since the scheduler can significantly reduce packet reordering at the receiver. Additionally, Kuhn et al. shows that DAPS can provide higher throughput and shorter application delay than RR. Furthermore Kuhn et al. identify some flaws of the DAPS scheduler. First, DAPS does not perform better than RR when paths are homogeneous, and may instead introduce additional overhead. They also point out that poor RTT estimations may significantly lower the efficiency of DAPS. This flaw may be especially important as the algorithm may be slow to react to scenarios where the RTTs of the paths change over time, since the scheduler does not renew its view on the path characteristics until the next batch of packets are scheduled.

Ferlin et al. [19] compared the performance of LowRTT, DAPS and OTIAS for MPTCP using an emulated network scenario, where a client and server exchange data over two paths with competing background traffic. They considered three distinct traffic patterns; video streaming over a constant bit rate, web traffic and bulk transfers. For bulk transfers, their results indicate that LowRTT and OTIAS are competitive in regards to goodput, while DAPS performs worse than the others. However, both OTIAS and DAPS provide lower receive buffer occupancy than LowRTT. For web traffic in lossless scenarios, all three schedulers performed similarly in regards to completion time. Both OTIAS and DAPS performed worse than the LowRTT scheduler in lossy scenarios as the total download size increases. For constant bit rate traffic, their results show that DAPS generally performs worse than the other schedulers in terms of application delay. Ferlin et al. note that both OTIAS and DAPS are flawed, arguing that their strategies to build up queues prevent them from reacting to changes in the paths quick enough, resulting in sub-optimal performance. Based on these observations, Ferlin et al. proposed the Blocking Estimation-based (BLEST) scheduler for MPTCP. As with OTIAS and DAPS, BLEST is a path scheduler aimed
at minimizing packet reordering. Unlike OTIAS and DAPS, however, BLEST bases its
decisions on an estimation of the amount of head-of-line blocking at the receiver, based
on MPTCP’s connection level flow control. This is done by estimating how long a packet
sent on a slow path occupies space in the receive window, and comparing it against how
many packets the faster path could send in the same amount of time. If this number of
packets does not fit in the receive window, sending on the slower path would result in
head-of-line blocking. Thus, BLEST can decide to not transmit on a path if it predicts
that sending on the path is not beneficial. Ferlin et al. evaluated BLEST against the
LowRTT scheduler and single-path TCP using the same experimental setup as for OTIAS
and DAPS, and validated the results using real-world experiments. Their results suggest
that BLEST is better equipped to avoid head-of-line blocking than the alternative schedulers.

Lim et al. [31] proposed the Earliest Completion First (ECF) scheduler, and evaluated
it in both an experimental testbed and real networks. Similarly to the LowRTT scheduler,
ECF prioritizes the fastest path in terms of RTT until its congestion window is filled.
Additionally, once the fastest path is blocked by its congestion control, ECF evaluates
whether it is actually beneficial to send on a slower path. As such, like BLEST, ECF
may decide to not send on a slower path, in order to wait for the faster path to recover.
ECF bases this decision on how much data there is left to send, along with path RTT
and capacity estimations. For their testbed experiments, Lim et al. used a mobile device
connected to a Wi-Fi access point and an LTE interface, which was communicating with
a server. The performance of ECF was then compared against the performance of the
LowRTT, DAPS and BLEST schedulers. For the testbed, they measured the performance
for both video streaming and web traffic. Their results show that ECF achieves a higher
average bit rate for video streaming traffic than the other schedulers, where BLEST slightly
outperforms LowRTT and DAPS performs worse than the other schedulers. According to
Lim et al., the reason why ECF outperforms the other schedulers is because ECF allocates
traffic close to the ideal, showing an overall higher utilization of the best path as opposed to the other schedulers. Their results also show that ECF is able to build the congestion window of the fastest path quicker than the other schedulers, while keeping it more stable, minimizing the amount of times the congestion control is set back to slow start. Lim et al. also showed that ECF is significantly more robust than the other schedulers against head-of-line blocking as paths become more asymmetric. For web downloads, ECF also compared favorably, generally performing as good or better than the other schedulers. For the real-world experiments, a similar set of experiments were made, however, this time over the Internet, and only by comparing ECF against LowRTT. These results corroborate the results gained in the experimental testbed, with ECF outperforming LowRTT.

Dreibholz et al. [11] introduced a stream scheduler which maps data from CMT-SCTP streams to specific paths in order to limit head-of-line blocking. They evaluated their scheduler against the RR path scheduler. Their evaluation was based on a simulation of a scenario where a server and client exchange data over two to three separate paths, and the effect of the different schedulers on bulk transfers over asymmetrical paths were investigated. This was done by varying the delay and bandwidth of one of the paths and the send/receiver buffers on the client and server. Their results show that both the RR and the stream mapping schedulers can achieve high throughput for symmetrical paths. The stream mapping scheduler outperformed RR when the paths were asymmetrical, likely due to RR experiencing excessive packet reordering. Dreibholz et al. also showed that RR provides all streams with similar throughput and delays, whereas the stream mapping scheduler provides higher throughput and lower latencies for streams sent over the best path when paths are dissimilar. Their results also show the stream mapping scheduler is beneficial for the overall throughput and delay in asymmetric scenarios.

Eklund et al. [13] proposed a Dynamic Stream-aware (DS) scheduler for CMT-SCTP.
The DS scheduler is a part path, part stream scheduler which takes path characteristics into consideration when selecting which streams to send. The scheduler prioritizes the path most beneficial for transmission by evaluating the transfer time required to send one message over the path, when taking path RTTs, congestion windows and congestion control state (slow start and congestion avoidance) into consideration. Next, the DS scheduler selects a stream to send on that path. The DS scheduler prioritizes streams that have most recently used the selected path first. The stream that was most recently sent on the selected path is always chosen first, provided it has data to send. This behaviour is intended to reduce reordering at the receiver when path characteristics differ. Eklund et al. evaluated the DS scheduler against the RR scheduler and the stream mapping scheduler proposed by Dreibholz et al. [11]. In order to investigate the impact of stream scheduling, they additionally compared the DS scheduler against a variant with the stream scheduling component disabled. They evaluated the schedulers using a simulated scenario where a multihomed sender sends small signaling messages over multiple streams to a multihomed receiver. In their evaluation they kept the characteristics of one path constant while increasing the asymmetry on the other path in terms of delay and packet losses. They then measured the average packet latency for each stream. Their results show that both the path and stream scheduling components of the DS scheduler can reduce the average packet delay for such traffic.

In [14], Eklund et al. extended the DS scheduler to take stream priority into consideration. They evaluated this variant using a similar set of experiments as described in the previous paragraph, except this time only against the RR scheduler and standard SCTP. Their results show that CMT-SCTP can reduce the latency for symmetrical and nearly symmetrical network scenarios. For more heterogeneous network scenarios, their results indicate that the DS scheduler can reduce the message latency compared to RR.
4 Experimental Validation of MPQUIC

Before evaluating packet schedulers for MPQUIC, we validated the MPQUIC installation and the experimental setup which were used for the evaluations (Sections 6 and 8), against the results achieved by De Coninck et al. [9]. In order to do this, experiments conducted by De Coninck et al. were replicated and the achieved results were compared against theirs. Specifically, the performance evaluation for large file downloads in [9] was replicated. The scenario involves a multihomed client requesting the transmission of a 20 MB file over a single HTTP stream. The time required for the download was then measured. In the evaluation, the performance of MPQUIC was compared against the performances of single-path TCP, MPTCP and single-path QUIC. The Linux kernel implementations of TCP and MPTCP [33] and the go implementations of QUIC [15] and MPQUIC [18] were used for the performance evaluation. For the sake of fairness, the file was transferred over HTTPS using TLS 1.2 for TCP and MPTCP and the built-in cryptography for QUIC and MPQUIC. For the single-path protocols, the CUBIC congestion control was used. For MPTCP and MPQUIC, the OLIA congestion control and the Lowest-RTT-First packet scheduler was used. In a web-scenario, we assume that the 0-RTT connection establishment provided by QUIC is a common occurrence. As such, handshake caching was enabled for QUIC and MPQUIC. In between each measurement, the server and client applications were restarted. Since the handshake cache is not permanently stored on the server, a new handshake cache needs to be generated whenever the server application is started. Thus, a dummy session was set up in order to set up the handshake cache before each measured run. Generic Receive Offload (GRO), Generic Segmentation Offload (GSO) and TCP Segmentation Offload (TSO) are optimizations which offloads the segmentation and reconstruction of data to the Network Interface Controller (NIC) instead of the CPU. This can improve performance at very high data rates. The performance improvements gained by using these features are not expected to be noticeable for the bandwidths used in this evaluation, however. Since these features may also interfere with the transmissions, GRO,
GSO and TSO were disabled.

De Coninck et al. measured the performance using the Mininet network emulator [43]. In their experiments, both the client and server were multihomed, with two available paths for the connection. In contrast, this work relies on measurements from a virtualized environment provided by Karlstad University, which consists of five separately emulated nodes and the virtual links between them. In this environment, only the client is multihomed, which we assume is more representative of a real-world scenario, such as the case with a smartphone. Additionally, De Coninck et al. set the maximal receive windows to 16 MB, whereas the maximal receive windows were unchanged from the default values in the experiments presented in this work. In the case of TCP and MPTCP, the default value for the maximal receive window size may vary due to TCP’s window scaling option, whereas QUIC and MPQUIC strictly uses 16 MB. This may accentuate the performance impact of packet reordering as the sender may become blocked by the flow control more often, but may also be more representative of a typical use-case scenario. We assume that most users would not set the protocol parameters to anything other than their defaults.

Figure 4.1 provides an illustration of the virtualized network scenario. The nodes in the virtualized network and their roles are:

- **Client**: The client node represents a device with two network interfaces, such as a smartphone. One interface is connected to a WLAN access point, whereas the other
interface is connected to a Mobile Broadband (MBB) access point. These access points are represented by separate nodes described below. The client node runs Debian GNU/Linux 9 kernel version 4.15.0 for the (MP)QUIC measurements. Since the kernel implementation of MPTCP did not run on this kernel version, version 4.9.65.3 for (MP)TCP was used instead. The discrepancy between the kernel versions may possibly have an impact on the measurements. Due to time constraints, we were not able to perform the measurements on the same kernel versions.

- **Server**: The server node is connected to the virtual network via the gateway node. The server is running Debian GNU/Linux 9 kernel version 4.15.0 for (MP)QUIC and 4.9.65.3 for (MP)TCP.
- **WLAN AP**: The WLAN AP node represents a Wi-Fi access point, which routes the traffic between the client and gateway nodes. The node is running Debian GNU/Linux 9 kernel version 4.15.0.
- **MBB AP**: This node acts similarly to the WLAN AP node, but represents the MBB access point. The node runs Debian GNU/Linux 9 kernel version 4.15.0.
- **Gateway**: The gateway node is connected with the server, the WLAN AP and the MBB AP nodes.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Low-BDP</th>
<th>High-BDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTT (ms)</td>
<td>0.1</td>
<td>100</td>
</tr>
<tr>
<td>Queuing delay (ms)</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>Bandwidth (Mbps)</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Loss (%)</td>
<td>0</td>
<td>2.5</td>
</tr>
</tbody>
</table>

The topology for the virtualized network was set up using static routing for the server, gateway, and access point nodes, and source routing for the client node. In [9], De Coninck et al. used an experimental design approach to vary the characteristics of the two paths. The measurements were carried out over four different classes of network scenarios:
• **Low-BDP-no-loss**: Environments with low bandwidth delay product without emulated random losses. Some losses may still occur naturally.
• **Low-BDP-loss**: Environments with low bandwidth delay product with emulated random losses.
• **High-BDP-no-loss**: Environments with high bandwidth delay product without emulated random losses. Some losses may still occur naturally.
• **High-BDP-loss**: Environments with high bandwidth delay product with emulated random losses.

Table 4.1 lists the parameter space used by De Coninck et al. to configure the two network paths for the different classes. For the lossless classes (Low-BDP-no-loss and High-BDP-no-loss), the row for the random losses shown in Table 4.1 are simply ignored, as the loss rate is set to zero. The exact configurations used for their measurements are available in [18, 17] and were thus used as a basis for the experiments conducted in this work. There are 253 different network configurations for each class. Additionally, the initial path used to establish the connection was varied for each network configuration, resulting in 506 scenarios per class. The file was transferred three times per protocol for each scenario.

### 4.1 Tools

A number of tools were required in order to provide the proper infrastructure for the evaluations.

**(MP)QUIC Client** The application (program) run by the client node for the (MP)QUIC experiments. This application connects to the server and requests a file over HTTP. The (MP)QUIC Client application measures the time between the transmission of the connection packet to the reception of the last byte of the requested file. The application is based on that which was used by De Coninck et al. [18]. A small change was made to the application and the interface to the MPQUIC protocol to allow a user to specify which path to use for
the initial path.

**(MP)QUIC Server** The application run by the server node for the (MP)QUIC experiments. This application transmits a file to the client over a single stream over the available paths. Similarly to the (MP)QUIC client, the same server application used by De Coninck et al. was used [18].

**(MP)TCP Server** The application run by the server node for the (MP)TCP experiments. This is an HTTPS (MP)TCP server application based on the application used by De Coninck et al. [16].

**Wget** This tool [21] was used as the (MP)TCP client application. The `time` command was used to measure the download completion time.

**Netem** The `netem` network emulator was used in order to emulate various properties of the links of the network, such as the delay and packet loss. The `rate` command was used to control the bandwidth of the links. The `delay` command was used to emulate propagation delays, and the `loss` command was used to introduce random losses. To emulate queuing delay, the queue sizes of the nodes were set to the bandwidth (queuing) delay product using the `limit` command. Netem was used on the interfaces of the WLAN and MBB nodes in order to control the delays experienced by both the client and server, and the outgoing bandwidths of the server. The outgoing bandwidths from the client side was set using netem for outgoing traffic on the interfaces of the client node. Netem was used to emulate the network conditions at the following interfaces: `wlan-to-gateway`, `wlan-to-client`, `mbb-to-gateway`, `mbb-to-client`, `client-to-wlan` and `client-to-mbb`.

**Tcpdump** Tcpdump was used for additional data collection at the interfaces of the
server and client nodes (server-to-gateway, client-to-wlan, client-to-mbb).

**Control scripts** Python control scripts were used in order to remotely setup the sce-

Control scripts Python control scripts were used in order to remotely setup the sce-

Control scripts Python control scripts were used in order to remotely setup the sce-

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the

4.2 Analysis

De Coninck et al. analyzed their results using two different metrics. The first metric is the
often than MPTCP. This in turn means that MPTCP is more likely to be receive window blocked than MPQUIC. Additionally, the ORP algorithm used by the Linux implementation of MPTCP (described in Section 3) may further hamper the performance of the best path by retransmitting packets from the slower path [9].

The performance differences for the low-BDP-loss and high-BDP-loss classes are shown in Figures 4.3 and 4.5. It appears that QUIC and MPQUIC tend to perform better in lossy environments than their TCP counterparts. The results are almost identical to those of De Coninck et al. [9]. De Coninck et al. attribute the differences in performance to the fact that the (MP)QUIC ACK frame is more capable of signaling packet losses than the TCP equivalence (SACKs), which suggests that MPQUIC does not suffer as much from head-of-line blocking.

Figure 4.4 shows that QUIC and MPQUIC perform better than TCP and MPTCP in a large fraction of the high-BDP scenarios without random losses. Although the time ratio metric for this specific class is missing in [9], meaning that no direct comparison can be made, this result can still be considered expected. According to De Coninck et al. [9], MPTCP is more susceptible to bufferbloat than MPQUIC because MPTCP tends to prefer the initial path as its congestion window has more time to grow, whereas MPQUIC allows a fairer growth of the congestion windows of all paths. Additionally, as previously mentioned, MPQUIC’s LowRTT scheduler is also able to maintain a closer estimation of path RTTs.

The second metric used by De Coninck et al. is the Experimental Aggregation Benefit (EAB) [9]. The EAB compares the goodput achieved by the multipath variant of a protocol against the sum of the goodputs achieved by the single path variant over each path. Equation 4.1 shows how the EAB is calculated, where $n$ is the number of paths, $G_m$ is the mean goodput of the multipath protocol, $G_i$ is the mean goodput measured for the single-path protocol on path $i$ and $G_{max}$ is the highest achieved goodput achieved by the single-path protocol on any path. An EAB equal to or greater than one means that the multipath protocol is able to fully aggregate the goodput over all paths, whereas an EAB under 0
means that the protocol does not benefit from multipath. An EAB of 0 indicates that the protocol does not benefit from multipath, but does not perform worse than its single-path variant, and is the ideal worst-case scenario. An EAB between 0 and 1 means that the
Figure 4.6: EAB of MPTCP and MPQUIC for lossless low-BDP scenarios. Both TCP and QUIC benefit from multipath.

Figure 4.7: EAB of MPTCP and MPQUIC for lossy low-BDP scenarios. QUIC still benefits from multipath in lossy environments, whereas multipath is not beneficial for TCP.

Figure 4.8: EAB of MPTCP and MPQUIC for lossless high-BDP scenarios. MPQUIC can still be advantageous in high-BDP scenarios.

Figure 4.9: EAB of MPTCP and MPQUIC for lossy high-BDP scenarios. QUIC and TCP’s abilities to take advantage of multiple paths are compromised in lossy high-BDP scenarios.

The protocol does benefit from using multiple paths, however, it does not fully aggregate all paths. The closer the EAB is to one (or above), the closer it achieves the ideal aggregation. An EAB of -1 means that the multipath protocol was not able to transmit data at all.
\[ EAB = \begin{cases} \frac{G_m - G_{max}}{(\sum_{i=1}^{s} G_i) - G_{max}}, & \text{if } G_m \geq G_{max}^s \\ \frac{G_m - G_{max}}{G_{max}^s}, & \text{otherwise} \end{cases} \] (4.1)

As shown in Figure 4.6, both MPTCP and MPQUIC are beneficial compared to their single-path variants for scenarios in the low-BDP-no-loss class. QUIC tends to benefit more from multipath than TCP. Upon introducing losses, MPQUIC generally remains beneficial, whereas multipath is not beneficial to TCP, as illustrated in Figure 4.7. These results correspond with the results in [9]. "Best path first" in these figures indicate that the path which achieved the highest goodput using the single-path protocol was used as the initial path for the multipath protocols. Unlike in [9], neither MPQUIC nor MPTCP appear to be particularly affected by the choice of initial path. Further investigation may be required to explain these results. For longer transfers, the initial handshake take up only a small fraction of the total transfer, and may thus have a limited effect on the total download time.

Figure 4.8 shows that MPQUIC can be advantageous in lossless high-BDP scenarios, whereas the benefit of MPTCP decreases. MPQUIC can still be beneficial in some lossy high-BDP scenarios, whereas MPTCP generally is not, as illustrated by Figure 4.9. As previously mentioned, MPQUIC is better equipped to handle losses than MPTCP.

4.3 Conclusion

The results gained from the validation of MPQUIC appear to coincide with the results in [9], despite some minor discrepancies. Thus, the correctness of the MPQUIC installation and experimental setup can be considered confirmed.
5 Design and Implementation of ECF

Previous related work (see Section 3) for packet schedulers for transport protocols such as MPTCP and CMT-SCTP has shown that proper scheduling is crucial for the performance when concurrently using multiple paths, in terms of both throughput and latency. As the goal of this work is to improve the performance of web-applications using MPQUIC, we thus see the packet scheduling process as a prime candidate for improvement.

In Section 4 it was shown that multipath can often be beneficial to QUIC when using the LowRTT scheduler, however, based on previous work, the assumption is that this approach is not optimal. A major concern with the LowRTT scheduler is that it does not handle heterogeneous paths well, as it may begin sending on a comparatively slow path as soon as the congestion windows of the faster paths have been filled. Such a scheduling decision can result in significant head-of-line blocking at the receiver, which can be detrimental to the overall transfer time. At the same time, we expect that MPQUIC will be commonly used for handheld devices such as smartphones in order to aggregate their WLAN and MBB interfaces. However, WLAN and MBB connections can differ wildly in characteristics. As such, path heterogeneity is a substantial problem which needs to be overcome. We seek to implement a packet scheduler for MPQUIC which addresses this concern. As presented in Section 3, experimental packet schedulers such as OTIAS, BLEST and ECF can all outperform the LowRTT scheduler for MPTCP. Of these, ECF appeared to perform best for heterogeneous paths. ECF not only takes path RTT into consideration, but also estimates the bandwidths of the paths via the congestion windows. Using this information, the ECF scheduler can make the decision to not send on a path if it is not considered beneficial, and instead waits for the congestion window of a faster path to recover. For this reason, the ECF scheduler was implemented for MPQUIC.
5.1 Overview of the ECF Scheduler

The ECF scheduler, like the LowRTT scheduler, prioritizes the path with the lowest RTT. Hereafter, this path is referred to as $p_f$. Thus, if $p_f$ is not blocked by the congestion control, $p_f$ can safely be chosen for transmission. In the case where $p_f$ is blocked, the packet may be scheduled on the best available path (second fastest path) instead, which is referred to as $p_s$. If ECF finds that transmission over $p_s$ is not beneficial, the scheduler will decide to not transmit and instead wait for $p_f$ to become available.

The pseudo code for the ECF scheduler is presented in Algorithm 5.1 [31]. Let $k$ be the current number of packets yet to be transmitted over all streams, and $rtt_f$, $rtt_s$, $cwnd_f$, $cwnd_s$, $\sigma_f$ and $\sigma_s$ be the RTTs, congestion windows and standard deviation of the RTTs for $p_f$ and $p_s$ respectively. In order to compensate for variabilities in the RTT and congestion window values, the margin value $\delta = max(\sigma_f, \sigma_s)$ is used when deciding which path is most beneficial. The constant $\beta = 0.25$ represents hysteresis which is introduced in order to prevent the scheduler from switching states too often.

The time required to send $k$ packets over $p_f$ and $p_s$ is approximately $\frac{k}{cwnd_f} \times rtt_f$ and $\frac{k}{cwnd_s} \times rtt_s$ respectively. If $p_f$ is blocked by its congestion window, the sender must wait approximately $rtt_f$ for $p_f$ to become available. Thus, it takes approximately $(1 + \frac{k}{cwnd_f}) \times rtt_f$ in order to wait for and transmit $k$ packets over $p_f$. If the inequality in Equation 5.1 is satisfied, it is more beneficial to wait for $p_f$ than to send over $p_s$. If ECF has previously decided to wait for $p_f$, hysteresis is additionally taken into consideration during the evaluation, as shown in Equation 5.2, which toggles between the two variants. Any path slower than $p_s$ is also expected to satisfy this inequality whenever $p_s$ does since its RTT is either greater than or equal to $rtt_s$, and does thus not need to be considered.

$$(1 + \frac{k}{cwnd_f}) \times rtt_f < rtt_s + \delta \quad (5.1)$$
\[(1 + \frac{k}{cwnd_f}) \times rtt_f < (1 + \text{waiting} \times \beta)(rtt_s + \delta) \quad (5.2)\]

When waiting for \(p_f\), it takes at least \(2rtt_f\) for \(p_f\) to begin transfer. ECF additionally makes another check to assure it is truly faster to wait for \(p_f\) than to transmit all data over \(p_s\) using the inequality shown in Equation 5.3. If the inequality is satisfied, it is not beneficial to send over \(p_s\).

\[\frac{k}{cwnd_s} \times rtt_s \geq 2rtt_f + \delta \quad (5.3)\]

As shown in Algorithm 5.1, the scheduling process can be split into two separate steps. First, the fastest path \(p_f\) and the fastest unblocked path \(p_s\) must be found. This can be done using an algorithm similar to that of LowRTT. The pseudocode for this step is shown in Algorithm 5.2. This step can then be combined with Algorithm 5.1.

### 5.2 Implementation

The ECF scheduler was implemented in the go implementation of MPQUIC [18], and is partially based on the implementation of the LowRTT scheduler for MPQUIC in go as well as the implementation of ECF for MPTCP in the Linux kernel by Lim et al. [31]. The source code for the ECF scheduler can be found in Appendix A. This section begins with a brief overview of the scheduling component in the MPQUIC implementation in go. Then, the implementation details of the ECF scheduler are presented.

#### 5.2.1 Overview of the Scheduling Component

An overview of how the scheduler component of the go implementation of MPQUIC works is as follows:

1. The scheduling component updates its packet history and flow controls, and checks for retransmissions.
Algorithm 5.1 ECF path scheduler [31]

Input:
- $p_f$, $p_s$: fastest and second fastest path, respectively
- $rtt_f$, $rtt_s$: RTTs of fastest and second fastest path
- $cwnd_f$, $cwnd_s$: congestion windows of fastest and second fastest path
- $\sigma_f$, $\sigma_s$: standard deviations of the RTTs for the fastest and second fastest path
- $k$: number of packets yet to be transmitted
- $waiting$: toggle between waiting and non-waiting states (default 0)
- $\beta$: hysteresis constant

Output:
- Path selected for transmission, or no transmission

Algorithm:

if $p_f$ is not blocked then
    return $p_f$
else
    $\delta = \max(\sigma_f, \sigma_s)$
    if $(1 + \frac{k}{cwnd_f}) \times rtt_f < (1 + waiting \times \beta)(rtt_s + \delta)$ then
        if $\frac{k}{cwnd_s} \times rtt_s \geq 2rtt_f + \delta$ then
            // Transmission not beneficial on $p_s$, wait for $p_f$
            $waiting = 1$
            return no transmission
        else
            return $p_s$
    else
        $waiting = 0$
        return $p_s$
    end if
end if

61
Algorithm 5.2 ECF - Selecting the fastest and second fastest paths

Input:
paths: set of available paths

Output:
Fastest path, $p_f$ (may be blocked)
Second fastest path, $p_s$ (unblocked)
RTTs of fastest and second fastest path, $rtt_f$ and $rtt_s$, respectively

Algorithm:
$rtt_f$ and $rtt_s$ are initiated as an arbitrarily large number
for each $p$ in paths do
    if $p.rtt > rtt_f$ then
        if $p.rtt < rtt_s$ then
            if $p$ is not blocked then
                $rtt_s = p.rtt$
                $p_s = p$
            end if
        end if
    end if
    $rtt_f = p.rtt$
    $p_f = p$
end for
2. The path scheduler selects a path. The initial path is only used for specific control messages at the beginning and end of a connection, and is never chosen unless it is the only path. The path scheduler may select a blocked path on the condition that there are retransmissions. If no path has been chosen, the sender may still send ACK frames on any path that is not the initial path.

3. Upon selecting a path, a new packet is composed. If there are frames carrying cryptographic information, such frames are prioritized and sent without control frames. If there are no frames carrying such information, control frames are bundled together with as many STREAM frames that can fit in the payload. The packet may then be encrypted and sent on the selected path.

4. The scheduling component duplicates the packet on paths with unknown RTTs, provided that it is not the initial path.

The implementation of the ECF scheduler primarily concerns step 2 in this process. Since step 4 can be beneficial to ECF by providing quicker RTT estimations of unknown performing paths, it was decided that this behavior shall be left intact.

5.2.2 Implementation of ECF

As previously mentioned, Algorithm 5.2, which is used to select $p_f$ and $p_s$, is similar to the LowRTT scheduler. Both schedulers look for the fastest paths in regards to RTT. Due to the similarities between the schedulers, this step of the ECF implementation is based on the previously existing implementation of the LowRTT scheduler. The LowRTT scheduler uses smoothed RTT estimations from the congestion control which are represented as 64-bit integer nanosecond counts. A path is only considered for sending if it is not blocked by the congestion control (blocked paths may still retransmit), if the path is not marked as potentially failed and if it is not the initial path. The LowRTT scheduler prioritizes paths with known RTTs. If all paths have unknown RTTs the least used path is prioritized. Finally, the scheduler finds an eligible path with the lowest RTT. This is done by comparing
the RTT of each path with the lowest previously encountered RTT. If there are no eligible paths, no path is chosen.

There are two main changes required for the ECF scheduler. First, ECF needs to find the best path regardless of whether it is blocked or not, whereas the LowRTT scheduler never considers blocked paths for sending. The required change is trivial; the check to see if a path is blocked can simply be removed, as this condition is examined later in Algorithm 5.1. Secondly, if the fastest path is blocked, ECF needs to find the fastest path which is not blocked. The required change is once again minimal. After comparing the RTT of a path with the lowest encountered RTT so far, the ECF scheduler additionally compares the RTT of the path with the second lowest RTT encountered so far, and updates the second lowest RTT accordingly on the condition that the path is not blocked.

In the second step of the scheduling process, shown in Algorithm 5.1, ECF decides whether the sender should send on $p_f$, wait for $p_f$, or send on $p_s$. $p_f$ is always selected provided it is not blocked by its congestion control or if there are pending retransmissions. Otherwise, ECF must assess whether it should wait for $p_f$ to become unblocked or send on $p_s$. If $p_f$ is blocked, and there is no path $p_s$ (i.e., all paths are blocked), no path is selected for transmission. If there is an unblocked path $p_s$, the algorithm proceeds to the final evaluation.

At this point ECF requires additional information from the congestion control, namely the congestion windows and RTT standard deviations of $p_f$ and $p_s$. However, MPQUIC operates using the mean deviation rather than the standard deviation. Since both the standard deviation and mean deviation are measures of variability, the mean deviation is used as replacement in this implementation. The mean deviation is more stable against outlying samples than the standard deviation, however, this also means that ECF may react slower against changing path characteristics. Congestion windows are represented as unsigned 64-bit byte counts and the mean deviations as 64-bit nanosecond counts. Furthermore, ECF must know how much data, $k$, is left to be transmitted. Streams
are represented by a stream data structure, which hold the dataForWriting field. The dataForWriting field contains the bytes queued for transmission on the stream. The total length of the dataForWriting fields of each stream combined thus represents the number of bytes yet to be transmitted. This value is represented as an unsigned 64-bit integer, and is used for $k$. Since $k, cwnd_f, cwnd_s, rtt_f, rtt_s$, and $\delta$ are all represented as 64-bit integers, no conversion between types are necessary. To avoid overflow, it is beneficial to keep these values as 64-bit integers. However since the inequalities in 5.2 and 5.3 require division, the equivalent inequalities in Equations 5.4 and 5.5 are used in their respective place, where $\theta = 1/\beta = 4$.

$$\theta(k + cwnd_f) \times rtt_f < cwnd_f(rtt_s + \delta)(\theta + waiting) \quad (5.4)$$

$$rtt_s \times k \geq cwnd_s(2rtt_f + \delta) \quad (5.5)$$
6 Evaluation of ECF

In order to evaluate the performance of the ECF scheduler for MPQUIC, its performance was compared against the performance of the default (LowRTT) scheduler in terms of throughput and latency. The evaluation was performed in two steps. First, we investigated the impact the schedulers have on the download time for the transfers of single files. This traffic pattern mainly requires high throughput, and was investigated in order to confirm the assumption that the performance of MPQUIC can be further optimized using more sophisticated scheduling decisions. Next, since QUIC is primarily designed for the web, the performance of the ECF scheduler for web traffic was evaluated. Web traffic results in a traffic pattern different from single file transfers. Web traffic primarily benefit from lower latencies as opposed to throughput, as is the case for the single file transfers. Web transfers typically consist of multiple streams which are dependent on each other, i.e. some HTTP requests can not be sent until a previous request has been processed. For example, a web browser must first begin to parse the root HTML file of a web page before it can begin to request resources such as CSS style sheets or JavaScripts, which in turn may be depended on by other resources. This means that requests do not reach the server at the same time, which may impact scheduling decisions.

6.1 Evaluation of ECF for Single File Transfers

Both LowRTT and ECF prioritize the fastest path in regards to RTT as long as it has space left in its congestion window. If file sizes are small enough that the congestion window of the fastest path is never filled, both schedulers are expected to exclusively send on the fastest path, regardless of the characteristics of the worst path. Only when the congestion window of the fastest path is exceeded, or if two or more paths have similar RTTs, are either scheduler expected to use more than one path. In the case where the fastest path becomes blocked by its congestion control, LowRTT will begin sending on a slower path, whereas
ECF may decide to wait for the faster path to become available again. The differences between the schedulers are thus expected to become more apparent as the path asymmetry and the file sizes increase. To evaluate the performance of ECF against LowRTT, a set of experiments were conducted where the time required to download files of various sizes using HTTP over increasing path asymmetry was measured.

6.1.1 Parameters

In this section, the parameters considered during the evaluation and their expected impact are covered.

**Path round-trip time:** The path round-trip time is two times the end-to-end delay of a path. In these evaluations the end-to-end delay comprises as follows:

- **Processing delay:** the time it takes for a router to process the header of a packet. This includes the time it takes to check for packet corruption as well as the time it takes to determine an appropriate output link.
- **Queuing delay:** the time a packet spends waiting in queues at the routers of the network. The queuing delay depends on the queue size and congestion level of the router.
- **Transmission delay:** the time it takes to push a packet into the link. This delay is dependent on the link bandwidth and the length of the packet.
- **Propagation delay:** the time it takes for the signal to propagate through the link. The propagation delay varies depending on the propagation speed over the link medium as well as the distance the signal needs to propagate.

**Impact:** Path RTT is critical to the performance of both LowRTT and ECF, as it is used as the primary factor for path selection.

**Link bandwidth:** The link bandwidth determines the maximum amount of bits which
can be outstanding on a link at once.

*Impact:* ECF takes bandwidth into consideration when selecting paths, whereas LowRTT does not, meaning that LowRTT may begin sending on non-beneficial paths.

**File size:** The size of the file transmitted from the server to the client.

*Impact:* The choice of file size impacts the download times. For small file sizes, both schedulers are expected to remain single path unless the paths are symmetrical, whereas differences in the schedulers are expected to manifest as the size increases.

**Initial path:** The initial path is the path used in order to establish a connection.

*Impact:* While the choice of initial path is meaningful to the total download time, it has no impact on scheduling decisions.

**Handshake information caching:** MPQUIC can cache information about the handshake of a previous connection with the server in order to provide 0-RTT handshakes in the future.

*Impact:* Enabling handshake information caching reduces the delay between the transmission of the connection packet and the first packet containing application data, but does not affect scheduling decisions.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>WLAN</th>
<th>MBB</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTT (ms)</td>
<td>20</td>
<td>20, 100, 200</td>
</tr>
<tr>
<td>Bandwidth (Mbps)</td>
<td>50</td>
<td>50, 10, 1</td>
</tr>
</tbody>
</table>

**6.1.2 Experimental Setup**

The measurements were performed on the virtualized network described in Section 4, with the difference that the client and server nodes were running Debian GNU/Linux 9 kernel
version 4.9.65.3. Table 6.1 displays the link parameters used in the evaluation. Since we are interested in investigating the impact of path asymmetry, the path characteristics over the WLAN were kept constant, while the characteristics of the path over MBB were varied. The range of file sizes were 64 KB, 128 KB, 256 KB, 512 KB and 1024 KB. The files were downloaded one at a time over a single stream. Between each download, the client and server applications were restarted. Each download was repeated 30 times for each network scenario, and the mean download times and the number of packets sent on each path were analyzed. In these measurements, the path over WLAN was used for the initial path, and handshake caching was enabled. Before each measurement, a dummy session was used to set up the handshake cache. The OLIA congestion control was used when evaluating both schedulers, and the maximal receive window was set to 16 MB. The initial congestion window was set to 32 packets. GSO and GRO were disabled. The tools used for the MPQUIC measurements detailed in Section 4.1 were reused for these evaluations, using a tweaked version of the control script to accommodate for the discrepancies in parameters and transport protocols.

6.1.3 Results

**MBB Bandwidth = 50 Mbps:** Figure 6.1 shows the download times for the scenarios where the bottleneck bandwidths of both paths are 50 Mbps. As expected, there is little difference between the performance of the two schedulers in these scenarios. Table 6.2 shows the percentage of packets sent by the server for each scheduler. For MBB = 50 Mbps, 20 ms the two paths are symmetrical. As such, it is not unexpected that both paths send close to 50% of the total packet count for either scheduler, as neither should have a clear preference for one or the other. However, ECF does show a larger number of packets sent on WLAN than LowRTT does. Due to natural variance in the samples, the perceived RTT of the two paths may differ slightly. In the case of the LowRTT, the perceived difference does not matter, as it simply moves on to the other path once the 'faster' path is congestion
window limited. ECF makes additional assessments taking congestion windows and the number of packets left to send into account in these scenarios, which may explain why ECF appears to schedule sending on WLAN more often than LowRTT. Regardless, the different scheduling decisions does not have a large impact on the total download times.

For MBB = 50 Mbps, 100 ms, ECF tends to be stricter against the slower path for transfers which complete after 100 ms. In these cases, ECF shows a preference for waiting on best path, whereas LowRTT starts to utilize the slower path. The different scheduling decisions may have some impact, as ECF performs slightly better than LowRTT for 256 KB and 512 KB transfers. Note that the number of packets sent on MBB increases as file sizes get smaller for both schedulers. The first packet scheduled on the slower path must be sent at the earliest after 100 ms, which is around the time the transfer finishes for 64 KB. Both schedulers duplicate packets on paths with unknown RTTs, so this result indicates that these packets are exclusively duplicated packets (which do not contain any new data) sent at the beginning of the connection. Since the total packet count for short transfers are rather small, duplicated packets may inflate the packet counts for MBB.

For MBB = 50 Mbps, 200 ms, the number of packets sent on WLAN appear to be similar for both schedulers. At this point both schedulers primarily select WLAN for transmission. Again, duplicated packets may inflate the number of packets sent on MBB, as the first packet sent on MBB arrives after 200 ms. 200 ms is also when the behavior of the schedulers diverge, as LowRTT show a larger number of packets sent on MBB than ECF for 1024 KB, which is the only transfer to exceed 200 ms. The decision to either wait for WLAN or send on MBB appear to have no noticeable effect on the total transfer time.

**MBB Bandwidth = 10 Mbps:** The download times for the slightly more asymmetrical scenarios, where the bandwidths are 50 Mbps for WLAN, and 10 Mbps for MBB are shown Figure 6.2. As the RTTs of the two paths become asymmetrical, the impact different scheduling decisions becomes more prominent. For MBB = 10 Mbps, 20 ms, there is no significant difference between the total download times for the two schedulers, and the
Figure 6.1: Mean download times against 95% confidence intervals in symmetric bandwidth scenarios. In symmetric bandwidth scenarios LowRTT and ECF generally perform similarly regardless of path RTTs.

Table 6.2: Percentage of packets sent on WLAN for symmetric scenarios in terms of bandwidth.

<table>
<thead>
<tr>
<th>MBB</th>
<th>File Size (KB)</th>
<th>ECF (%)</th>
<th>LowRTT (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 Mbps, 20 ms</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>58.6</td>
<td>54.4</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>52.7</td>
<td>51.0</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>55.5</td>
<td>53.8</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>50.1</td>
<td>48.4</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>52.5</td>
<td>51.8</td>
<td></td>
</tr>
<tr>
<td>50 Mbps, 100 ms</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>88.4</td>
<td>89.2</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>92.1</td>
<td>92.6</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>95.6</td>
<td>84.8</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>97.0</td>
<td>89.6</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>97.9</td>
<td>91.0</td>
<td></td>
</tr>
<tr>
<td>50 Mbps, 200 ms</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>87.5</td>
<td>87.9</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>92.9</td>
<td>92.8</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>96.1</td>
<td>95.7</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>98.3</td>
<td>98.3</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>98.2</td>
<td>94.6</td>
<td></td>
</tr>
</tbody>
</table>

number of packets sent on each path, shown in Table 6.3, corroborate that the schedulers operate similarly in these scenarios. Due to the differences in bandwidth, the path over WLAN is able to have more outstanding packets at a time in comparison to the path over MBB, which thus results in a larger number of packets sent on WLAN. As with MBB = 50

71
Mbps, 20 ms, ECF shows a tendency to wait for WLAN as opposed to sending on MBB for smaller file sizes.

![Image of Figure 6.2](image-url)

**Figure 6.2:** Mean download times against 95% confidence intervals in asymmetric bandwidth scenarios. ECF outperforms LowRTT as path asymmetry and file sizes increase.

Table 6.3: Percentage of packets sent on WLAN for asymmetric scenarios in terms of bandwidth.

<table>
<thead>
<tr>
<th>MBB</th>
<th>File Size (KB)</th>
<th>ECF (%)</th>
<th>LowRTT (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 Mbps, 20 ms</td>
<td>64</td>
<td>61.6</td>
<td>57.4</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>69.1</td>
<td>68.7</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>66.9</td>
<td>66.3</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>70.4</td>
<td>70.0</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>77.5</td>
<td>77.0</td>
</tr>
<tr>
<td>10 Mbps, 100 ms</td>
<td>64</td>
<td>88.1</td>
<td>89.6</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>92.6</td>
<td>93.4</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>95.1</td>
<td>82.8</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>96.9</td>
<td>89.6</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>98.0</td>
<td>88.3</td>
</tr>
<tr>
<td>10 Mbps, 200 ms</td>
<td>64</td>
<td>88.4</td>
<td>87.0</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>93.0</td>
<td>92.9</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>95.7</td>
<td>95.9</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>98.1</td>
<td>98.3</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>98.1</td>
<td>94.7</td>
</tr>
</tbody>
</table>

For MBB = 10 Mbps, 100 ms, the differences between the two schedulers become apparent as the download completion times exceed 100 ms. For file sizes of 256 KB, 512
Figure 6.3: Mean download times against 95% confidence intervals in highly asymmetric bandwidth scenarios. ECF greatly outperforms LowRTT at high path asymmetry.

Table 6.4: Percentage of packets sent on WLAN for highly asymmetric scenarios in terms of bandwidth.

<table>
<thead>
<tr>
<th>MBB</th>
<th>File Size (KB)</th>
<th>ECF (%)</th>
<th>LowRTT (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Mbps, 20 ms</td>
<td>64</td>
<td>74.7</td>
<td>75.0</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>78.3</td>
<td>73.9</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>81.2</td>
<td>78.9</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>88.3</td>
<td>78.0</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>93.5</td>
<td>86.5</td>
</tr>
<tr>
<td>1 Mbps, 100 ms</td>
<td>64</td>
<td>88.8</td>
<td>88.6</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>92.3</td>
<td>92.5</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>95.8</td>
<td>83.9</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>96.8</td>
<td>89.3</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>97.8</td>
<td>89.9</td>
</tr>
<tr>
<td>1 Mbps, 200 ms</td>
<td>64</td>
<td>88.7</td>
<td>88.5</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>93.3</td>
<td>93.2</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>95.7</td>
<td>95.3</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>98.4</td>
<td>98.1</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>98.3</td>
<td>94.6</td>
</tr>
</tbody>
</table>

KB and 1024 KB, ECF shows a pronounced improvement over the LowRTT scheduler. In these scenarios, ECF prefers to stay single-path whereas the LowRTT scheduler begins to send on a non-beneficial path. The scenario where MBB = 10 Mbps, 200 ms, shows similar results. As the time required for a transfer exceeds the RTT for MBB, ECF outperforms LowRTT. Again, the performance differences are mainly noticeable for the larger file sizes.
**MBB Bandwidth = 1 Mbps:** Figure 6.3 and Table 6.4 show the results for the most asymmetric scenarios. ECF greatly outperforms the LowRTT scheduler in such scenarios. The results show that multipath is not beneficial and should stay single-path as path asymmetry increases. While ECF is more suited to decide when to stay single-path, it is also apparent that even ECF may make poor scheduling decisions when RTTs are equal. This is because ECF assumes that sending on the path with the lowest RTT is always beneficial, which may not necessarily be true. Path RTTs vary naturally; sending on a path may lead to queue buildup somewhere along the path, which temporarily increases the queuing delay of the path. As such, the most used path may be unnecessarily punished the longer it is used. Path characteristics other than RTT, such as bandwidth, can also impact how beneficial a path is, but are not considered by ECF during the initial selection of the fastest path.

### 6.2 Evaluation of ECF for Web Traffic

The evaluations covered so far (Section 4 and 6.1) have been focused on measuring the performance on transfers for single files. Since our primary goal is to improve the performance of the web, we also evaluated the performance of ECF for web-like traffic. Web page loading is inherently different from single file transfers. Since a web browser may only request new resources as it encounters them during the loading process, pauses between requests naturally occur, resulting in a more fragmented transfer as opposed to single file transfers. Web traffic is also typically comprised of several streams. To investigate the impact ECF has on such a traffic pattern, another set of experiments was conducted comparing the performance of ECF and LowRTT. For these experiments, a similar setup to that described in Section 6.1 was used. Instead of transferring files of various sizes, we mirrored five different web pages on our server, which comprise various total download sizes and number of elements. Table 6.5 shows the selected web pages.

To emulate the page loading process of a web browser, a program (see Appendix D)
was created which can replay the process of loading web pages using dependency graphs\textsuperscript{2} from the WProf project [46]. The dependency graphs contain information about when the web request and computing activities for each resource start in relation to the activities of other resources. The replay program reads the dependency graph, and requests the root HTML file, which may then begin to be processed. Once certain criteria are met, the program may begin requesting and process other resources according to the information in the dependency graphs. The time required to download and process all resources of the page, the \textit{Page Load Time} (PLT), was then measured for each web page over the network scenarios inferred from Table 6.1. Each page was downloaded 30 times for each network scenario, and the mean PLTs and the number of packets sent on each path were analyzed. HTTP/2’s server push feature was not enabled during these experiments.

\subsection*{6.2.1 Results}

\textbf{MBB Bandwidth = 50 Mbps:} Figure 6.4 and Table 6.6 shows the PLTs and the number of packets sent on each path for the three most symmetric network scenarios in terms of bandwidth. For MBB = 50 Mbps, 20 ms and MBB = 50 Mbps, 100 ms, there is no significant difference between the performance of ECF and LowRTT. For MBB = 50 Mbps, 200 ms, however, LowRTT outperforms ECF for Dropbox. A reason for this may be because ECF is operating on imperfect information. ECF bases its decision to wait for the faster path partially on how much data there is left to send on the connection. However, ECF

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|}
\hline
\textbf{Web page} & \textbf{Resources} & \textbf{Total size (KB)} \\
\hline
bing.com & 2 & 52 \\
wikipedia.org & 19 & 176 \\
youtube.com & 25 & 480 \\
ask.com & 24 & 1392 \\
dropbox.com & 35 & 1900 \\
\hline
\end{tabular}
\caption{Web pages used during the evaluation.}
\end{table}

\textsuperscript{2}Dependency graphs and web resources are available at \url{http://wprof.cs.washington.edu/spdy/tool} [45].

75
Figure 6.4: Mean PLT against 95% confidence intervals in symmetric bandwidth scenarios. Imperfect information about the total transfer size may negatively impact the performance of ECF.

Table 6.6: Percentage of packets sent on WLAN for symmetric scenarios in terms of bandwidth.

<table>
<thead>
<tr>
<th>MBB</th>
<th>Web page</th>
<th>ECF (%)</th>
<th>LowRTT (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 Mbps, 20 ms</td>
<td>bing.com</td>
<td>57.6</td>
<td>53.9</td>
</tr>
<tr>
<td></td>
<td>wikipedia.org</td>
<td>52.2</td>
<td>52.5</td>
</tr>
<tr>
<td></td>
<td>youtube.com</td>
<td>51.7</td>
<td>50.5</td>
</tr>
<tr>
<td></td>
<td>ask.com</td>
<td>50.1</td>
<td>48.6</td>
</tr>
<tr>
<td></td>
<td>dropbox.com</td>
<td>48.3</td>
<td>51.9</td>
</tr>
<tr>
<td>50 Mbps, 100 ms</td>
<td>bing.com</td>
<td>90.0</td>
<td>89.7</td>
</tr>
<tr>
<td></td>
<td>wikipedia.org</td>
<td>96.7</td>
<td>96.7</td>
</tr>
<tr>
<td></td>
<td>youtube.com</td>
<td>80.0</td>
<td>73.5</td>
</tr>
<tr>
<td></td>
<td>ask.com</td>
<td>75.4</td>
<td>70.0</td>
</tr>
<tr>
<td></td>
<td>dropbox.com</td>
<td>77.6</td>
<td>57.9</td>
</tr>
<tr>
<td>50 Mbps, 200 ms</td>
<td>bing.com</td>
<td>91.4</td>
<td>93.1</td>
</tr>
<tr>
<td></td>
<td>wikipedia.org</td>
<td>96.3</td>
<td>96.5</td>
</tr>
<tr>
<td></td>
<td>youtube.com</td>
<td>88.3</td>
<td>82.0</td>
</tr>
<tr>
<td></td>
<td>ask.com</td>
<td>87.8</td>
<td>74.2</td>
</tr>
<tr>
<td></td>
<td>dropbox.com</td>
<td>86.8</td>
<td>63.0</td>
</tr>
</tbody>
</table>

is only aware of the number of bytes that has been requested so far, and not the transfer size of resources from pending requests. Because ECF does not have information about the total transfer size, it may begin to make poor scheduling decisions, i.e. waiting for the faster path when in reality it would be beneficial to send on the slower path, or vice
Figure 6.5: Mean PLT against 95% confidence intervals in asymmetric bandwidth scenarios. The results hint at possible performance issues when using ECF even as path asymmetry increases.

Table 6.7: Percentage of packets sent on WLAN for asymmetric scenarios in terms of bandwidth.

<table>
<thead>
<tr>
<th>MBB</th>
<th>Web page</th>
<th>ECF (%)</th>
<th>LowRTT (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 Mbps, 20 ms</td>
<td>bing.com</td>
<td>55.4</td>
<td>58.5</td>
</tr>
<tr>
<td></td>
<td>wikipedia.org</td>
<td>52.5</td>
<td>52.6</td>
</tr>
<tr>
<td></td>
<td>youtube.com</td>
<td>52.1</td>
<td>54.3</td>
</tr>
<tr>
<td></td>
<td>ask.com</td>
<td>55.5</td>
<td>53.1</td>
</tr>
<tr>
<td></td>
<td>dropbox.com</td>
<td>56.5</td>
<td>54.9</td>
</tr>
<tr>
<td>10 Mbps, 100 ms</td>
<td>bing.com</td>
<td>89.8</td>
<td>90.9</td>
</tr>
<tr>
<td></td>
<td>wikipedia.org</td>
<td>97.1</td>
<td>97.1</td>
</tr>
<tr>
<td></td>
<td>youtube.com</td>
<td>82.5</td>
<td>73.1</td>
</tr>
<tr>
<td></td>
<td>ask.com</td>
<td>77.5</td>
<td>71.5</td>
</tr>
<tr>
<td></td>
<td>dropbox.com</td>
<td>77.7</td>
<td>62.0</td>
</tr>
<tr>
<td>10 Mbps, 200 ms</td>
<td>bing.com</td>
<td>91.0</td>
<td>90.2</td>
</tr>
<tr>
<td></td>
<td>wikipedia.org</td>
<td>97.3</td>
<td>96.8</td>
</tr>
<tr>
<td></td>
<td>youtube.com</td>
<td>89.1</td>
<td>80.1</td>
</tr>
<tr>
<td></td>
<td>ask.com</td>
<td>89.3</td>
<td>73.7</td>
</tr>
<tr>
<td></td>
<td>dropbox.com</td>
<td>85.9</td>
<td>63.7</td>
</tr>
</tbody>
</table>

versa. Such a phenomenon becomes more likely as the resource dependencies become more complex.

**MBB Bandwidth = 10 Mbps:** Although the differences between LowRTT and ECF are not statistically significant, the results corroborate that ECF’s preference to wait on the
best path may be detrimental when the scheduler is working with incomplete information about the total transfer. Figure 6.5 and Table 6.7 show the PLTs and the number of packets sent on each path for each web page respectively.

Figure 6.6: Mean PLT against 95% confidence intervals in highly asymmetric bandwidth scenarios. ECF outperforms LowRTT.

Table 6.8: Percentage of packets sent on WLAN for highly asymmetric scenarios in terms of bandwidth.

<table>
<thead>
<tr>
<th>MBB</th>
<th>Web page</th>
<th>ECF (%)</th>
<th>LowRTT (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Mbps, 20 ms</td>
<td>bing.com</td>
<td>76.4</td>
<td>78.4</td>
</tr>
<tr>
<td></td>
<td>wikipedia.org</td>
<td>78.4</td>
<td>78.1</td>
</tr>
<tr>
<td></td>
<td>youtube.com</td>
<td>75.5</td>
<td>69.7</td>
</tr>
<tr>
<td></td>
<td>ask.com</td>
<td>83.3</td>
<td>73.8</td>
</tr>
<tr>
<td></td>
<td>dropbox.com</td>
<td>82.2</td>
<td>73.9</td>
</tr>
<tr>
<td>1 Mbps, 100 ms</td>
<td>bing.com</td>
<td>89.6</td>
<td>88.7</td>
</tr>
<tr>
<td></td>
<td>wikipedia.org</td>
<td>96.3</td>
<td>96.3</td>
</tr>
<tr>
<td></td>
<td>youtube.com</td>
<td>85.5</td>
<td>79.7</td>
</tr>
<tr>
<td></td>
<td>ask.com</td>
<td>90.5</td>
<td>78.6</td>
</tr>
<tr>
<td></td>
<td>dropbox.com</td>
<td>85.6</td>
<td>71.8</td>
</tr>
<tr>
<td>1 Mbps, 200 ms</td>
<td>bing.com</td>
<td>89.9</td>
<td>90.4</td>
</tr>
<tr>
<td></td>
<td>wikipedia.org</td>
<td>96.0</td>
<td>96.1</td>
</tr>
<tr>
<td></td>
<td>youtube.com</td>
<td>89.3</td>
<td>81.5</td>
</tr>
<tr>
<td></td>
<td>ask.com</td>
<td>91.7</td>
<td>76.4</td>
</tr>
<tr>
<td></td>
<td>dropbox.com</td>
<td>88.5</td>
<td>72.8</td>
</tr>
</tbody>
</table>
**MBB Bandwidth = 1 Mbps**: As shown in Figure 6.6, ECF outperforms LowRTT in all three network scenarios as web page complexity increases. The performance issues regarding ECF described in the previous paragraphs appear to manifest when the scheduler decides to wait for the faster path, while it is actually beneficial to send on the slower path. However, in these scenarios, the path asymmetry is so high that it is generally never beneficial to utilize the slower path. Thus, ECF’s decision to primarily stay single path, as shown in Table 6.8, reduces the overall time to load the web pages.

### 6.3 Conclusion

The results of this evaluation show that the LowRTT scheduler is not optimal; in some instances, ECF can reduce the average download time of a single file transfer by up to $1/4$ of the time required by LowRTT. However, ECF is not optimal either. In some cases, ECF may wrongly assume that the path with the lowest RTT is the most beneficial path, which is not necessarily the case. Furthermore, ECF may make poor scheduling decisions if it is working with imperfect information, as may be the case for more complex traffic patterns such as web traffic. This issue could be somewhat alleviated using HTTP/2’s server push feature; by predicting which resources are needed by the client before receiving the actual requests, the server quicker can get a more accurate reading of the total transfer size, which would allow ECF to make better decisions.
7 A Stream Priority Scheduler for MPQUIC

Sections 5 and 6 detail the design, implementation and evaluation of a sophisticated path scheduler for MPQUIC, which can significantly reduce the page load times of web pages compared to the default path scheduler. Notably, it was shown that performance of the scheduler is negatively impacted in scenarios where it lacks information about the transfer. In such scenarios, the scheduler is more likely to make poor scheduling decisions. To further improve the performance of web pages, the possibility of supplying the packet scheduler with additional information from the application layer was investigated. This was done by exploring the stream concept of (MP)QUIC. The MPQUIC implementation in go uses a simple Round Robin based scheduler when bundling STREAM frames into packets, ensuring fairness to all streams of the connection. However, this behaviour may not be optimal for web traffic. In Section 6, the total page load time was used as a metric to show the performance disparity between path schedulers for web traffic. However, a web browser may begin rendering a web page before all resources have been fetched. The goal is to provide a better user experience by making the page interactive as soon as possible, instead of simply showing a blank page until the entire page has loaded.

The critical rendering path is the sequence of steps a web browser needs to perform before it can begin rendering a web page [23]. These steps require the browser to evaluate the HTML, CSS and JavaScript which make up the web page. The critical rendering path begins by constructing the Document Object Model (DOM) tree from the HTML and the CSS Object Model (CSSOM) tree from any potential CSS files [23]. The DOM and CSSOM trees are then combined into the final rendering tree used to render the page. Some JavaScripts can modify the DOM and CSSOM trees, and may as such also be required in order to properly render the page [23]. Note that other types of resources, such as images, are not required to begin rendering the page. By prioritizing the sending of streams which make up the critical rendering path, the time to render may be reduced. As such, we introduce a priority based stream scheduler for MPQUIC, which can take additional information
provided by the application in order to make more informed scheduling decisions.

7.1 Design

The goal of the Stream Priority Scheduler is to allow a sender to prioritize the sending of some streams over others, with a primary focus on the web. Since (MP)QUIC does not have knowledge of what is contained in the application data, the application should be responsible for determining which streams should be prioritized, and (MP)QUIC should simply schedule the sending of streams accordingly. When deciding which stream to send on, the stream priority scheduler looks for the highest priority stream which is not blocked by its flow control. If there are several unblocked streams with the same priority, the scheduler makes a selection from that set. Streams are iterated over in a Round Robin fashion to ensure fairness for streams with the same priority. The pseudo code for the stream priority scheduler is displayed in Algorithm 7.1.

7.2 Implementation

The current MPQUIC implementation lacks support for stream prioritization. Thus, the priority field was added to the stream data structure, which is used to represent streams. A stream priority is represented as an unsigned 8-bit integer, allowing up to 256 different traffic classes, which is expected to provide well enough granularity in a web context.

Stream scheduling and path scheduling can be considered to be separate processes. Stream scheduling takes place during step 3 in the overview of the packet scheduling process provided in Section 5.2.1. Thus, the stream scheduler was implemented independent from the path scheduler. This choice allows users to combine path and stream schedulers as they see fit. Furthermore, the selection of a stream can also be seen as an independent process from the actual bundling of STREAM frames. The default stream scheduler does not make a stream selection, and simply iterates over the different streams in an RR fashion at frame bundling time. To implement a priority based stream scheduler, the scheduling
Algorithm 7.1 Stream Priority Scheduler

Input:
- streams: set of open streams
- priorities: set of stream priorities in descending order, based on user input
- startIndex: start index

Output:
- Stream selected for transmission, or no transmission

Algorithm:

for each p in priorities do
    i = 0
    while i < len(streams) do
        s = streams[(i + startIndex) % len(streams)]
        if s.priority == p then
            if s is not blocked then
                startIndex = i
            end if
        end if
        i +=
    end while
end for
// All streams blocked, nothing to transmit
return no transmission
component was extended to additionally allow for stream selection. The frame bundling process also required modification to allow it to prioritize the selected stream. From a long term perspective, these modifications allow future stream schedulers to select streams using other criteria, while reusing the same frame bundling process. The modified scheduling process is as follows:

1. The scheduling component updates its packet history and flow controls, and checks for retransmissions.
2. The path scheduler selects which path should be used to send the packet.
3. The priority stream scheduler selects a stream for transmission based on its priority according to Algorithm 7.1. If the default stream scheduler is used, no stream is selected.
4. STREAM frames are bundled into the packet payload. The frame bundling process prioritizes sending on the selected stream. If no stream has been selected in the previous step, frames are bundled using the default process. The packet may then be sent.
5. The scheduling component duplicates the packet on paths with unknown RTTs.

The source code for the Priority Stream scheduler can be found in Appendix B.
8 Evaluation of the Stream Priority Scheduler

The Stream Priority scheduler is designed to reduce the time required to download web objects depending on their priority. As such, this evaluation only focuses on the impact of the schedulers for web traffic. The evaluation was carried out in two steps. First, we repeat the experiments detailed in Section 6.2, this time comparing MPQUIC’s default stream scheduler against the Stream Priority scheduler. The ECF path scheduler was used in combination with both stream schedulers. Instead of measuring the PLT, the time to complete the critical rendering path (the time required to evaluate the HTML and CSS files) was measured. From hereon out, this time will be referred to as the ‘rendering time’. Additionally, the download time for each resource was measured in order to show how the scheduling decisions impact individual resources. The stream priorities were set on the sender side based on the requested resource type. Since we are interested in improving the rendering time, the critical rendering resources, namely HTML, CSS and JavaScripts, were prioritized. HTML and CSS are the primary resources which make up the critical rendering path, and were as such set to the highest priority. JavaScript can block the construction of the DOM and CSSOM, unless the HTML specifically states that the script can run asynchronously. Since there is no way to distinguish between synchronous and asynchronous scripts from the sender side, we settle with a middle ground, giving all scripts a medium priority. Table 8.1 shows a summary of how the sending resources were prioritized.

The dependency graphs used in these experiments may imply some implicit order of data transmission, which may possibly counteract MPQUIC’s scheduling decisions. The dependency graphs only take the start times of activities in relation to the corresponding start/end times of other resources into consideration when deciding when to start an activity. This means that, for example, resources which start the evaluation before the whole resource has finished downloading does this after a set amount of time after the first chunk of the response has been loaded. In such scenarios, the computation time is not based on the actual progress of the download. The computation time stated in the dependency graphs may as
such imply some level of link characteristics and/or sender optimizations from the time the dependency graphs were recorded, and our stream prioritizations has little impact on the computation time of the resource. As such, a second evaluation was performed. In the second evaluation, the previous experiment was repeated, however, this time, all web objects were requested concurrently (with no regards to dependencies between the resources) after receiving the root HTML. The time required to download all critical resources was then measured. This approach is not ideal either, since such traffic patterns are not necessarily representative of actual web traffic. However, it allows for closer monitoring of the effects of the stream schedulers in the absence of proper web browsers which implement MPQUIC.

Table 8.1: Stream priorities.

<table>
<thead>
<tr>
<th>Resource type</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML</td>
<td>255 (High)</td>
</tr>
<tr>
<td>CSS</td>
<td>255 (High)</td>
</tr>
<tr>
<td>JavaScript</td>
<td>250 (Medium)</td>
</tr>
<tr>
<td>Other</td>
<td>0 (Low)</td>
</tr>
</tbody>
</table>

8.1 Replaying Web Traffic

To evaluate the Stream Priority scheduler for web-like traffic, the experiments detailed in Section 6.2 were repeated, however, this time comparing different stream scheduling techniques. Since stream scheduling primarily affects the order in which data is sent, the total transfer time for web pages are not expected to be significantly affected. As such, this evaluation focuses on the page render time and per-resource download times. It bears to be mentioned, however, that the PLT could be reduced for some web pages by sending resources which other resources are dependent on earlier. In such scenarios, the PLT may correlate with the page render time, and is not considered at this stage. The following discusses the results of the experiments.
8.1.1 Results

**MBB Bandwidth = 50 Mbps:** As seen in Figure 8.1, the Stream Priority scheduler may actually perform worse than the default scheduler in regards to the render time as the size and complexity of the web pages increase. The Stream Priority scheduler can compete with the default stream scheduler up to a point, as shown in Figure 8.2, and may even perform slightly better for smaller pages. The reason is likely because the smaller pages have fewer dependencies, which reduces the risk of a poor scheduling decision. It appears that the performance of the Stream Priority scheduler may suffer as dependencies become more complex, due to sub-optimal priority configuration. For example, it may be more beneficial to prioritize JavaScript above all else in some cases, since JavaScripts can block the construction of the DOM and CSSOM. However, the stream priorities were not configured for such behavior in this evaluation. The individual download times for the most critical resources (HTML, CSS) for all web pages, shown in Figure 8.3, displays that the Stream Priority scheduler remains competitive in regards to the per-resource completion time. The reason that there is not a greater difference between the schedulers in this case is likely because the web pages may already be optimized for the critical rendering path; the root HTML may be laid out in such a way that the HTML parser encounters supplementary HTML and CSS files before other resources, allowing these resources to be requested earlier. Another contributing factor is that supplementary HTML and CSS files tend to be fairly small in size, ensuring low transfer times even as the sending is interleaved with less critical resources when using the default stream scheduler.

**MBB Bandwidth = 10 Mbps:** The results are similar to the previous scenarios even as the paths become more heterogeneous; the web pages do not benefit from stream priority in terms of render time, as shown in Figures 8.4 and 8.5. Figure 8.6 shows, however, that the Stream Priority scheduler does perform as well as the default stream scheduler for the prioritized resources.
MBB Bandwidth = 1 Mbps: Figures 8.7, 8.8 and 8.9 display similar results to previous scenarios; rendering time may increase using the Stream Priority scheduler, but the transfer times for the prioritized resources are not negatively affected. The performance issues likely lie in the configuration of stream priorities, and not the concept of stream prioritization in itself. The Stream Priority scheduler, as it has been evaluated here, is simply too naïve to meet the challenges posed by traffic with complex dependencies. Reducing rendering times via stream prioritization thus requires more information about the transfer on the sender side than what has been provided in this evaluation.

8.2 Independent Web Requests

The following evaluation focuses on a simpler traffic model where the client initially requests the root HTML of a web page, and upon completion requests all other resources in parallel regardless of dependency. The equivalent experiments to those described in Sections 6.2 and 8.1 were then performed, and the completion time for the critical rendering resources (HTML, CSS) were measured. The traffic evaluated in this step is not necessarily representative of the typical web case, but does highlight the effect of stream prioritization.

8.2.1 Results

MBB Bandwidth = 50 Mbps: Figure 8.10 shows the mean time required to download all critical rendering resources for each web page. Figure 8.11 shows the CDF of the time required to download these resources for all web pages combined. Figure 8.12 shows the individual download times of the highest prioritized resources (HTML and CSS files) for every page. As can be seen, stream prioritization has a positive effect on the completion time of the critical resources. The effect appears to be most apparent for complex web pages in terms of the number of resources and the total download size. This is expected; only as the resources have to compete for space in the congestion and/or receive windows
Figure 8.1: Mean render times against 95% confidence intervals in symmetric bandwidth scenarios. Prioritizing streams yielded a longer render time for complex web pages.

Figure 8.2: Render times for all web pages in symmetric bandwidth scenarios. The Stream Priority scheduler is generally outperformed by the default stream scheduler, but can benefit web pages with shorter critical rendering paths.

Figure 8.3: Per-resource download times for the prioritized resources for all five web pages in symmetric bandwidth scenarios. The Stream Priority scheduler and the default scheduler perform similarly in terms of the per-resource download time for the prioritized resources.
Figure 8.4: Mean render times against 95% confidence intervals in asymmetric bandwidth scenarios. The Stream Priority scheduler performs worse than the default stream scheduler for complex web pages.

Figure 8.5: Render times for all web pages in asymmetric bandwidth scenarios. The Stream Priority scheduler may perform similarly to, or worse, than the default stream scheduler.

Figure 8.6: Per-resource download times for the prioritized resources for all five web pages in asymmetric bandwidth scenarios. The two schedulers achieve similar download times for the prioritized resources in moderately asymmetric path scenarios.
WLAN = 50Mbps 20ms RTT, MBB = 1Mbps

RTT MBB = 20ms
RTT MBB = 100ms
RTT MBB = 200ms

Figure 8.7: Mean render times against 95% confidence intervals in highly asymmetric bandwidth scenarios. Even as path asymmetry increases, the Stream Priority scheduler is outperformed by the default scheduler.

WLAN = 50Mbps 20ms, MBB = 1Mbps

Figure 8.8: Render times for all web pages in highly asymmetric bandwidth scenarios. The Stream Priority scheduler performs better for smaller transfers, but is overtaken by the default scheduler as the page size increases.

WLAN = 50Mbps 20ms, MBB = 1Mbps

Figure 8.9: Per-resource download times for the prioritized resources for all five web pages in highly asymmetric bandwidth scenarios. The two schedulers display similar download times for the prioritized resources.
does stream prioritization have a substantial effect. For small web pages, the congestion and connection level receive windows are likely large enough to not pose a problem. As long a stream is never blocked for sending, the difference between sending first or in a round robin fashion is likely not noticeable, since the stream does not have to wait long regardless. As web pages become larger and more complex, the likelihood that resources must wait for the congestion or flow controls increase.

**MBB Bandwidth = 10 Mbps:** Figure 8.13 shows a similar difference between the two schedulers as the path asymmetry increases; there is only an improvement for larger web pages. However, as demonstrated in Figure 8.14, The Stream Priority scheduler achieves greater download time reductions compared to the default stream scheduler than in the more symmetric scenarios. By ensuring that critical rendering resources are sent first, prioritized streams are less likely to be impacted by the path asymmetry.

**MBB Bandwidth = 1 Mbps:** As the paths become more asymmetric, the effect of stream prioritization becomes more pronounced. Figure 8.16 shows that web pages that did not benefit from stream priorities in previous scenarios may benefit as the path asymmetry increases. Stream priority still has little effect on smaller web pages. Figure 8.17, likewise, displays greater performance benefits for highly asymmetric paths. Figure 8.18 corroborate these findings; the Stream Priority scheduler shows a clear improvement for the download times of critical resources. As the path asymmetry increases, the effect of head-of-line blocking becomes more significant, due to the large differences in path characteristics. Stream prioritization increases the likelihood that the selected streams are sent over the same same (fastest) path, which leads to a lower amount of head-of-line blocking.
Figure 8.10: Mean completion times for the critical rendering resources against 95% confidence intervals in symmetric bandwidth scenarios. The Stream Priority scheduler can reduce the time required to download critical resources for transfers with a large amount of concurrent requests.

Figure 8.11: Completion times of critical rendering resources for all web pages in symmetric bandwidth scenarios. The Stream Priority scheduler outperforms the default stream scheduler.

Figure 8.12: Per-resource download times for the prioritized resources for all five web pages in symmetric bandwidth scenarios. The Steam Priority scheduler performs better than default scheduler.
Figure 8.13: Mean completion times for the critical rendering resources against 95% confidence intervals in asymmetric bandwidth scenarios. Prioritizing streams can reduce the download time of critical resources even as path asymmetry increases.

Figure 8.14: Completion times of critical rendering resources for all web pages in asymmetric bandwidth scenarios. The Stream Priority scheduler copes better with increasing path asymmetry.

Figure 8.15: Per-resource download times for the prioritized resources for all five web pages in asymmetric bandwidth scenarios. The Stream Priority scheduler outperforms the default stream scheduler.
WLAN = 50Mbps 20ms RTT, MBB = 1Mbps

Figure 8.16: Mean completion times for the critical rendering resources against 95% confidence intervals in highly asymmetric bandwidth scenarios. The effect of stream priorities become more pronounced as path asymmetry increases.

WLAN = 50Mbps 20ms, MBB = 1Mbps

Figure 8.17: Completion times of critical rendering resources for all web pages in highly asymmetric bandwidth scenarios. The Stream Priority scheduler outperforms the default stream scheduler.

WLAN = 50Mbps 20ms, MBB = 1Mbps

Figure 8.18: Per-resource download times for the prioritized resources for all five web pages in highly asymmetric bandwidth scenarios. The time to download larger critical resources is reduced using the Stream Priority scheduler.
8.3 Conclusion

In this section we have shown that stream prioritization can lower the time required to transfer select resources. The effect is mainly noticeable when a large amount of resources are requested concurrently. Stream prioritization may have the opposite of the intended effect, and can increase render times if the priorities are not configured properly. Prioritization of streams work well when the traffic is easily predictable. However, we found that for complex traffic patterns, more sophisticated prioritization of individual streams are required.
9 Discussion

Through the evaluations detailed in Sections 6 and 8, it was shown that packet scheduling can be crucial for the performance of web-like applications. This was done by implementing and evaluating path and stream schedulers for MPQUIC. The discussion is separated into two parts. First, the impact of path scheduling is discussed. The second part discusses the findings from the evaluation of stream schedulers.

9.1 Path Scheduling

The results presented in Section 6 show that sophisticated path selection is a requirement for multipath transport protocols. One of the main problems with multipath communication is path asymmetry. A poor scheduling decision can be very costly; due to re-ordering, a single packet sent on a slow path can significantly hold up the completion of a transfer. Ideally, a path scheduler should utilize multiple paths only when it is beneficial for the total transfer time, and switch to sending on the best path exclusively when this is no longer the case.

In Section 6, it was shown that ECF, a path scheduler capable of making the decision to not send on paths it does not deem beneficial, can handle path asymmetry well. ECF achieved greater performance than MPQUIC’s default scheduler for asymmetric network paths. The results also show that the ECF scheduler generally handles symmetrical network scenarios just as well as the default scheduler. However, the results also indicate there is room for improvement. The ECF algorithm comes in two parts. First, ECF always prioritizes the fastest path, defined as the path with the lowest smoothed RTT, as long as it is not blocked by the congestion control. Secondly, if the fastest path is not available for sending, ECF evaluates whether it is more beneficial to send on the second fastest path, or to wait for the fastest path to become available again. When selecting the fastest path, ECF does not take path characteristics other than path RTT into consideration. This can lead to some ambiguity when path RTT’s are similar, leading to sub-optimal scheduling.
decisions. Although we do not have a direct comparison between MPQUIC with ECF and single-path QUIC, the results suggest that ECF is not capable of performing as well as single-path QUIC in such scenarios. Taking a wider array of path characteristics into consideration during the selection of the fastest path, rather than just the path RTTs, could possibly alleviate the issue.

Furthermore, ECF may lose its edge against the default scheduler in scenarios involving complex traffic patterns. Likely, this is because of the lack of information about the entire transfer on the sender side. Staggered web requests may lead to slight performance issues for symmetrical to moderately symmetrical paths. The ECF scheduler may in some cases prefer to wait for the faster path, when in reality, it would have been more beneficial to send on a slower path. This is because the sender is not able to predict the total transfer size, due to the intermittency of the requests. More sophisticated server configurations could, for example, correlate web requests which are usually sent in conjunction, and use that information to gain a more accurate view of the entire transfer.

Looking at the results in Section 4, the performance of the default scheduler for MPQUIC may be greatly impacted when introducing random packet losses. The ECF and default path schedulers share similarities, and neither scheduler takes path loss rates into consideration when scheduling packets. As such, it is not unlikely that the performance of ECF may also be impacted by such conditions. However, due to time constraints, random losses have not been considered in the network scenarios used for the evaluations. Further investigation is required to discover how ECF performs in such adverse conditions.

9.2 Stream Prioritization

Proper prioritization of streams is not a trivial task. The results presented in Section 8 show that there is potential in the stream priority concept, as the download times for prioritized resources can be noticeably reduced for all the considered network scenarios. However, the stream priorities must be configured properly, as sub-optimal stream priorities may
result in longer load times for web pages, as shown in Section 8.1. The results indicate that
the main problem for the sender is the inability to predict which resources are the most
beneficial for the receiver. Many things can play into the optimal order of transmission. In
the evaluations covered in Section 8, we statically prioritized the sending of the most critical
resources used to render a web page. However, this method resulted in poor performance for
complex traffic patterns, as we found that the rendering times of the web pages increased
rather than decreased. Likely, this is because the prioritization strategy did not properly
consider the complexity of the web page loading process. This highlights a problem with the
Stream Priority scheduler; it requires careful configuration in order to perform satisfactory.

Ideally, one should not only take the resource type into consideration when scheduling
streams for web traffic, but also the dependencies between streams. This requires additional
information on the sender side. With enough time and resources, the optimal order to send
resources could be found for an individual web page through thorough testing. We do not
expect such an endeavor to be realistic in a majority of scenarios. As noted in Section 9.1,
senders could benefit by correlating web-requests that are usually requested together in
order to glean a more accurate view of the total transfer. This idea could be extended
to also note the order and timing of requests. This information could then be used to
infer the dependencies between resources, and to set the priority of streams accordingly.
However, such an approach may also not be realistic. Additionally, if the optimal order to
send resources is already known, stream scheduling on the transport layer may become a
non-concern, since a server could simply push the resources as they become needed by the
receiver. We thus seek a more generalized solution. In the context of web applications, it is
generally more likely that the receiver knows more about the optimal order data should be
received than the sender. As such, the receiver could give the sender feedback regarding
which data streams should be prioritized at any given time. HTTP/2 already allows clients
to set the priority and define the dependencies of each stream. MPQUIC could use this
information to dynamically predict the optimal sending order on a per-connection level.
10 Conclusion and Future Work

In this thesis, methods to reduce the time required to load web pages by the means of packet scheduling have been investigated. Through an experimental validation, it was shown that the transport protocol QUIC already shows great potential for reducing download times compared to TCP, and that QUIC’s recent multipath extension, MPQUIC offers additional performance benefits. However, the multipath approach requires careful scheduling. We have shown that injecting packets on slower network paths when faster paths are available can noticeably degrade the performance. An important potential use-case for MPQUIC is just that; web traffic over heterogeneous network paths. However, the default path scheduler for MPQUIC does not properly consider the characteristics of paths, and may perform poorly when the path qualities diverge. Taking inspiration from previous work on path schedulers for MPTCP, we implemented the ECF path scheduler, which is specifically designed to address such scenarios. Our evaluations show that sophisticated path scheduling can significantly increase the throughput and achieve lower latencies compared to the default path scheduler in homogeneous network scenarios.

Additionally, the possibilities of (MP)QUIC’s stream concept were explored. A Stream Priority scheduler was implemented for MPQUIC, which aims to reduce the download times of select web resources. By prioritizing certain streams, we have shown that the download times for individual resources can be significantly reduced. We also identified weaknesses of this approach; the act of setting appropriate priorities is not trivial and an improper configuration may even be detrimental to MPQUIC’s performance.

10.1 Future Work

Based on the results of our evaluations, following classes of future work have been identified:

- Further evaluation of existing packet schedulers against a wider array of network environments, including scenarios with random packet losses, competing traffic, changing
path characteristics and real world scenarios.

- This work has primarily focused on MPQUIC’s performance for end-to-end, web-like traffic. Future work may investigate the impact of scheduling decisions for other potential use-cases. For example, MPQUIC could be used by network operators to tunnel traffic using MPQUIC proxies in their network. This would allow single-homed end-points to benefit from the aggregation of multiple paths, while also allowing the network operators to have greater control over how the traffic is scheduled over their network.

- More robust path scheduling. While the ECF scheduler performs well, the evaluations presented in this work indicate that the ECF scheduler is still not ideal. As such, there is still work to be done in regards to path scheduling.

- Advanced stream scheduling. In Section 9.2, we identify the need for more advanced stream scheduling options. Appropriating the stream priority concept in HTTP/2 presents one such opportunity, as does other ways to relay stream dependency information to MPQUIC. The additional information gained by closer interaction with the application layer may significantly improve the MPQUIC’s stream scheduling decisions.
References


A ECF Source Code

This appendix includes the source code for the Earliest Completion First path scheduler for MPQUIC.

Listing A.1: Source code for the ECF path scheduler.

```go
func (sch *scheduler) selectPathEarliestCompletionFirst(s *session,
    hasRetransmission bool, hasStreamRetransmission bool, fromPth *path)
    *path {
    if len(s.paths) <= 1 {
        if !hasRetransmission && !s.paths[protocol.InitialPathID].SendingAllowed() {
            return nil
        }
        return s.paths[protocol.InitialPathID]
    }

    if hasRetransmission && hasStreamRetransmission && fromPth.rttStats.SmoothedRTT() == 0 {
        currentQuota := sch.quotas[fromPth.pathID]
        for pathID, pth := range s.paths {
            if pathID == protocol.InitialPathID || pathID == fromPth.pathID {
                continue
            }
            if sch.quotas[pathID] < currentQuota {
                return pth
            }
        }
    }

    var bestPath *path
    var secondBestPath *path
```
var lowerRTT time.Duration
var currentRTT time.Duration
var secondLowerRTT time.Duration
bestPathID := protocol.PathID(255)

pathLoop:
for pathID, pth := range s.paths {
    if pth.potentiallyFailed.Get() {
        continue pathLoop
    }

    if pathID == protocol.InitialPathID {
        continue pathLoop
    }

    currentRTT = pth.rttStats.SmoothedRTT()

    if lowerRTT != 0 && currentRTT == 0 {
        continue pathLoop
    }

    // Case if we have multiple paths unprobed
    if currentRTT == 0 {
        currentQuota, ok := sch.quotas[pathID]
        if !ok {
            sch.quotas[pathID] = 0
            currentQuota = 0
        }
        lowerQuota, _ := sch.quotas[bestPathID]
        if bestPath != nil && currentQuota > lowerQuota {
            continue pathLoop
        }
    }
if currentRTT >= lowerRTT {
    if (secondLowerRTT == 0 || currentRTT < secondLowerRTT) && pth.SendingAllowed() {
        // Update second best available path
        secondLowerRTT = currentRTT
        secondBestPath = pth
    }
}

if currentRTT != 0 && lowerRTT != 0 && bestPath != nil {
    continue pathLoop
}

// Update best path
lowerRTT = currentRTT
bestPath = pth
bestPathID = pathID
}

if bestPath == nil {
    if secondBestPath != nil {
        return secondBestPath
    }
    return nil
}

if hasRetransmission || bestPath.SendingAllowed() {
    return bestPath
}

if secondBestPath == nil {
    return nil
}
```go
var queueSize uint64

getQueueSize := func(s *stream) (bool, error) {
    if s != nil {
        queueSize = queueSize + uint64(s.lenOfDataForWriting())
    }

    return true, nil
}

s.streamsMap.Iterate(getQueueSize)

cwndBest := uint64(bestPath.GetCongestionWindow())
cwndSecond := uint64(secondBestPath.GetCongestionWindow())
deviationBest := uint64(bestPath.rttStats.MeanDeviation())
deviationSecond := uint64(secondBestPath.rttStats.MeanDeviation())

delta := deviationBest
if deviationBest < deviationSecond {
    delta = deviationSecond
}

xBest := queueSize
if queueSize < cwndBest {
    xBest = cwndBest
}

lhs := uint64(lowerRTT) * (xBest + cwndBest)
rhs := cwndBest * (uint64(secondLowerRTT) + delta)
if beta * lhs < (beta * rhs + sch.waiting * rhs) {
    xSecond := queueSize
    if queueSize < cwndSecond {
        xSecond = cwndSecond
    }
```
lhsSecond := uint64(secondLowerRTT) * xSecond
rhsSecond := cwndSecond * (2 * uint64(lowerRTT) + delta)

if lhsSecond > rhsSecond {
    sch.waiting = 1
    return nil
}

else {
    sch.waiting = 0
}

return secondBestPath
B Stream Priority Scheduler Source Code

This appendix includes the source code for the Stream Priority stream scheduler for MPQUIC.

B.1 Stream Selector

Listing B.1: Source code for the Stream Priority stream scheduler.

```go
func (sch *scheduler) selectStreamPriority(s *session) *stream {
    selectedPriority := protocol.StreamPriority(0)
    var streamToSend *stream

    streams, order := s.streamsMap.GetStreamPriorities()
    priorityLoop:
        for _, priority := range order {
            for _, stream := range streams[priority] {
                sw, _ := s.flowControlManager.SendWindowSize(stream.streamID)
                if stream.streamID != 1 && stream.streamID != 3 && !stream.finishedWriteAndSentFin() && sw != 0 {
                    selectedPriority = priority
                    break priorityLoop
                }
            }
        }

    fn := func(strm *stream) (bool, error) {
        if strm.streamID == 1 || strm.streamID == 3 {
            return true, nil
        }
        if strm.priority != selectedPriority {
            return true, nil
        }
```
B.2 Stream Frame Bundling

This following contains the code which is responsible for bundling frames into a packet. If no stream has been selected, frames are bundled in a simple round robin fashion. Otherwise, sending on the given stream is prioritized.

Listing B.2: Source code for the modified frame bundling process.

```go
func (f *streamFramer) maybePopNormalFrames(maxBytes protocol.ByteCount, strm *stream, pth *path) (res []*wire.StreamFrame) {
    frame := &wire.StreamFrame{DataLenPresent: true}
    var currentLen protocol.ByteCount

    fn := func(s *stream) (bool, error) {
        if s == nil || s.streamID == 1 {
            return true, nil
        }
    }
```
frame.StreamID = s.streamID
frame.Offset = s.writeOffset
frameHeaderBytes, _ := frame.MinLength(protocol.VersionWhatever)
if currentLen+frameHeaderBytes > maxBytes {
    return false, nil
}
maxLen := maxBytes - currentLen - frameHeaderBytes

var sendWindowSize protocol.ByteCount
lenStreamData := s.lenOfDataForWriting()
if lenStreamData != 0 {
    sendWindowSize, _ = f.flowControlManager.SendWindowSize(s.streamID)
    maxLen = utils.MinByteCount(maxLen, sendWindowSize)
}

if maxLen == 0 {
    return true, nil
}

var data []byte
if lenStreamData != 0 {
    data = s.getDataForWriting(maxLen)
}

shouldSendFin := s.shouldSendFin()
if data == nil && !shouldSendFin {
    return true, nil
}

if shouldSendFin {
    frame.FinBit = true
    s.sentFin()
frame.Data = data
f.flowControlManager.AddBytesSent(s.streamID, protocol.ByteCount(len(data)))

if f.flowControlManager.RemainingConnectionWindowSize() == 0 {
    f.blockedFrameQueue = append(f.blockedFrameQueue, &wireBlockedFrame{StreamID: 0})
} else if !frame.FinBit && sendWindowSize-frame.DataLen() == 0 {
    f.blockedFrameQueue = append(f.blockedFrameQueue, &wireBlockedFrame{StreamID: s.StreamID()})
}

res = append(res, frame)
currentLen += frameHeaderBytes + frame.DataLen()

if currentLen == maxBytes {
    return false, nil
}

frame = &wire.StreamFrame{DataLenPresent: true}
return true, nil

if strm == nil {
    f.streamsMap.RoundRobinIterate(fn)
} else {
    f.streamsMap.PrioritizedRoundRobinIterate(fn, strm.streamID)
}
return
C Control Script

This appendix includes the source code for the python script used to control the experiments used for the validations. Since the different evaluations required tweaking, various variations of the control script were created. For brevity, only one variation is shown here. This particular version was used for the validation of MPQUIC (Section 4).

Listing C.1: Source code for the control script.

```python
#!/usr/bin/python

from __future__ import print_function
import numpy as np
import math
import subprocess
import time
import resultsFormatter as rf
import datetime
import topos
import sys
import getopt

# ETC
USAGE = 'Usage: experimental-design.py -c <class> -p <protocol>
SERVER_APP = 'server_main'
CLIENT_APP = 'main'
CERT_PATH = 'go/src/github.com/lucas-clemente/quic-go/example/
SET_CLIENT_ROUTES = 'sudo ./route-script &
FILE_NAME = 'random'
CLIENT_LOG = 'output.log'
MTU_ETHERNET = 1500 * 8

# Username for SSH
USERNAME = 'alexrabi100'
```
# IP addresses used for remote commands via SSH

SSH_CLIENT_IP = '192.168.60.107'
SSH_SERVER_IP = '192.168.60.168'
SSH_MBB_IP = '192.168.60.169'
SSH_WLAN_IP = '192.168.60.156'
SSH_GW_IP = '192.168.60.135'

# The IP addresses for the server and client

CLIENT_WLAN_IP = '10.0.4.2'
CLIENT_MBB_IP = '10.0.5.2'
SERVER_IP = '10.0.0.2'
SERVER_PORT = '6121'

# Interface names

WLAN_CLIENT_I = 'wlan-client'
WLAN_GW_I = 'wlan-gw'
MBB_CLIENT_I = 'mbb-client'
MBB_GW_I = 'mbb-gw'
SERVER_GW_I = 'server-gw'
CLIENT_WLAN_I = 'wlan0'
CLIENT_MBB_I = 'mbb0'

# Tcpdump settings

TCPDUMP_SNAPLEN = '217' # bytes

# Timeout value (prevent infinite loops)

TIMEOUT = 3600 # no run should take longer than 1h

def getQueueSize(queueDelay, rate, mtu=MTU_ETHERNET):
    q = (float(queueDelay) * (float(rate) * 1024 * 1024))/mtu
    return max(math.ceil(q), 2)

def generateTcpdumpString(device, filename, snaplen = TCPDUMP_SNAPLEN):
return 'sudo tcpdump -i ' + device + ' -w ' + filename + ' -s ' + snaplen + ' udp'

def generateNetemDelCmd(device):
    return 'sudo tc qdisc del dev ' + device + ' root & '

def generateNetemAddCmd(device, rate, delay, loss, limit=1000):
    return 'sudo tc qdisc add dev ' + device + ' root netem rate ' + str(rate) + 'mbit delay ' + str(delay) + 'ms loss ' + str(loss) + ' limit ' + str(limit) + ' & '

def setupClient(bind_address, server_address, server_port, file_to_get):
    cmds = './ ' + CLIENT_APP + ' -b ' + bind_address + ' -c ' + 'https:// ' + server_address + ':' + server_port + '/' + file_to_get
    subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_CLIENT_IP, SET_CLIENT_ROUTES]).wait()
    subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_CLIENT_IP, cmds]).wait()

def startClient(bind_address, server_address, server_port, file_to_get, handshake_cache, protocol):
    cmds = './ ' + CLIENT_APP + ' -b ' + bind_address
    if protocol == 'MPQUIC':
        cmds = cmds + ' -m'
    if handshake_cache:
        cmds = cmds + ' -c '
    cmds = cmds + 'https:// ' + server_address + ':' + server_port + '/' + file_to_get + ' &> ' + CLIENT_LOG
    print(cmds)

    try:
        subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_CLIENT_IP, cmds]).wait(timeout = TIMEOUT)
return True

except subprocess.TimeoutExpired:
    print('Warning: Timeout expired!')
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_CLIENT_IP, 'pkill ' + CLIENT_APP]).wait()
return False

def startServer(server_port):
    cmds = './' + SERVER_APP + ' -www -certpath ' + CERT_PATH + ' -bind ' + '0.0.0.0:' + server_port
    subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_SERVER_IP, cmds]).wait()
time.sleep(1)

def setLinks(wlan, mbb):
    # Create netem strings
    wlan_client_netem_del = generateNetemDelCmd(WLAN_CLIENT_I)
    wlan_gw_netem_del = generateNetemDelCmd(WLAN_GW_I)
    mbb_client_netem_del = generateNetemDelCmd(MBB_CLIENT_I)
    mbb_gw_netem_del = generateNetemDelCmd(MBB_GW_I)
    wlan0_netem_del = generateNetemDelCmd(CLIENT_WLAN_I)
    mbb0_netem_del = generateNetemDelCmd(CLIENT_MBB_I)

    wlan_client_netem_add = generateNetemAddCmd(WLAN_CLIENT_I, wlan['bandwidth'], wlan['delay'], wlan['loss'], getQueueSize(wlan['queuing delay'], wlan['bandwidth']))
    wlan_gw_netem_add = generateNetemAddCmd(WLAN_GW_I, 0, wlan['delay'], wlan['loss'], getQueueSize(wlan['queuing delay'], wlan['bandwidth']))
    mbb_client_netem_add = generateNetemAddCmd(MBB_CLIENT_I, mbb['bandwidth'], mbb['delay'], mbb['loss'], getQueueSize(mbb['queuing delay'], mbb['bandwidth']))
    mbb_gw_netem_add = generateNetemAddCmd(MBB_GW_I, 0, mbb['delay'], mbb['loss'], getQueueSize(mbb['queuing delay'], mbb['bandwidth']))
wlan0_netem_add = generateNetemAddCmd(CLIENT_WLAN_I, wlan['bandwidth'], 0, 0, getQueueSize(wlan['queuing delay'], wlan['bandwidth']))
mbb0_netem_add = generateNetemAddCmd(CLIENT_MBB_I, mbb['bandwidth'], 0, 0, getQueueSize(mbb['queuing delay'], mbb['bandwidth']))

# Delete previous rules
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_WLAN_IP, wlan_client_netem_del]).wait()
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_WLAN_IP, wlan_gw_netem_del]).wait()
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_MBB_IP, mbb_client_netem_del]).wait()
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_MBB_IP, mbb_gw_netem_add]).wait()
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_CLIENT_IP, wlan0_netem_del]).wait()
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_CLIENT_IP, mbb0_netem_del]).wait()

# Add new rules
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_WLAN_IP, wlan_client_netem_add + wlan_gw_netem_add]).wait()
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_MBB_IP, mbb_client_netem_add + mbb_gw_netem_add]).wait()
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_CLIENT_IP, wlan0_netem_add + mbb0_netem_add]).wait()

def runTest(testId, wlan, mbb, scheduler, handshake_cache, file_size, filenames, protocol):
    print('Starting run ' + str(testId))
    dump_server_name = 'results/run' + str(testId) + '-' + filenames + '-'
dump_server = generateTcpdumpString(device = SERVER_GW_I, filename = dump_server_name)
dump_client_wlan = generateTcpdumpString(device = CLIENT_WLAN_I, filename = dump_wlan_name)
dump_client_mbb = generateTcpdumpString(device = CLIENT_MBB_I, filename = dump_mbb_name)

# Setup links using netem
setLinks(wlan, mbb)

# Generate download file and start server
subprocess.Popen(["ssh", "-q", USERNAME + '@' + SSH_SERVER_IP, 'dd if=/dev/zero of=' + FILE_NAME + ' bs=1M count=' + str(file_size)]).wait()
startServer(SERVER_PORT)

# Setup handshake information cache for client
client_ip = CLIENT_WLAN_IP
if mbb["init_path"] is 1:
    client_ip = CLIENT_MBB_IP
setupClient(client_ip, SERVER_IP, SERVER_PORT, '404')
time.sleep(5) # Drain packets from setup connection

# Start tcpdump
subprocess.Popen(["ssh", "-q", USERNAME + '@' + SSH_SERVER_IP, dump_server])
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_CLIENT_IP, dump_client_wlan])
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_CLIENT_IP, dump_client_mbb])

`time.sleep(1)` # Make sure tcpdump is up

# Start client (for real this time)
ok = startClient(client_ip, SERVER_IP, SERVER_PORT, FILE_NAME, handshake_cache, protocol)

`time.sleep(2)`

# Stop server and remove download file
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_SERVER_IP, 'pkill ' + SERVER_APP]).wait()
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_SERVER_IP, 'rm ' + FILE_NAME]).wait()

# Remove cache files from client
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_CLIENT_IP, 'rm cache_*']).wait()

# Stop tcpdumps
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_SERVER_IP, 'sudo pkill -f tcpdump']).wait()
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_CLIENT_IP, 'sudo pkill -f tcpdump']).wait()

# Gzip
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_SERVER_IP, 'gzip -f ' + dump_server_name]).wait()
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_CLIENT_IP, 'gzip -f ' + dump_wlan_name]).wait()
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_CLIENT_IP, 'gzip -f ' + dump_mbb_name]).wait()
# Retry if timed out
if not ok:
    print('Retry!')
return runTest(testId, wlan, mbb, scheduler, handshake_cache,
               file_size, filenames, protocol)

# Get results from client
subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_CLIENT_IP, 'tail
-1 ' + CLIENT_LOG + ' &>> expDesign/' + filenames + '.out'])

p = subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_CLIENT_IP,
                       'tail -1 ' + CLIENT_LOG], stdout=subprocess.PIPE)
out = p.communicate()[0]
print('Done!')

# Check correctness of output
try:
    return str(out, 'utf-8').strip().replace('
', '').split(' ')[2]
except:
    print('Retry!')
    return runTest(testId, wlan, mbb, scheduler, handshake_cache,
                    file_size, filenames, protocol)

def runTests(protocol, simulation_class, wlan, mbb, times=30, scheduler='Lowest-RTT-First', handshake_cache=1, file_size=20):
    results = []

    filenames = protocol+'-'+simulation_class+'-'+wlan['id']+str(wlan['init_path'])+str(wlan['bandwidth'])+str(wlan['delay'])+str(wlan['queuing delay'])+str(wlan['loss']).split('%')

    filenames = filenames+'-'+mbb['id']+str(mbb['init_path'])+str(mbb['bandwidth'])+str(mbb['delay'])+str(mbb['queuing delay'])+str(mbb['loss']).split('%')

    # Run tests multiple times
    for _ in range(times):
        result = runTests(protocol, simulation_class, wlan, mbb, times, scheduler, handshake_cache, file_size)
        results.append(result)

    return result
delay']]+'-'+str(mbb['loss']).split('%')[0]
output_file = filenames + '.json'

start_time = '{0:%a %b %d %X UTC %Y}'.format(datetime.datetime.now())
for i in range(0, times):
    result = runTest(i, wlan, mbb, scheduler, handshake_cache, file_size, filenames, protocol)
    results.append(result)

end_time = '{0:%a %b %d %X UTC %Y}'.format(datetime.datetime.now())

# Get metadata
p = subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_CLIENT_IP, 'uname -a'], stdout=subprocess.PIPE)
client_meta = str(p.communicate()[0], 'utf-8').strip().replace('
', '')

p = subprocess.Popen(['ssh', '-q', USERNAME + '@' + SSH_SERVER_IP, 'uname -a'], stdout=subprocess.PIPE)
server_meta = str(p.communicate()[0], 'utf-8').strip().replace('
', '')

rf.constructOutput(protocol, simulation_class, results, wlan, mbb, start_time, end_time, scheduler, handshake_cache, file_size, client_meta, server_meta, output_file)

def runTestSuite(simulation_class, protocol):
    exp_topos = []
    if simulation_class == 'low-BDP-no-loss':
        exp_topos = topos.getLowBDPNoLoss()
    elif simulation_class == 'low-BDP-loss':
        exp_topos = topos.getLowBDPLoss()
    elif simulation_class == 'high-BDP-no-loss':
exp_topos = topos.getHighBDPNoLoss()

elif simulation_class == 'high-BDP-loss':
    exp_topos = topos.getHighBDPLoss()

else:
    raise NameError('No such class')

if protocol not in ('MPQUIC', 'QUIC'):
    raise NameError('Protocol not supported')

# Translate topos to new format
parsed_topos = []

for topo in exp_topos:
    path1 = {'id': 'wlan', 'init_path': 1, 'bandwidth': topo['paths'][0]['bandwidth'],
             'delay': topo['paths'][0]['delay'],
             'queuing delay': topo['paths'][0]['queuingDelay'],
             'loss': topo['netem'][0][2].split()[1]}
    path2 = {'id': 'mbb', 'init_path': 0, 'bandwidth': topo['paths'][1]['bandwidth'],
             'delay': topo['paths'][1]['delay'],
             'queuing delay': topo['paths'][1]['queuingDelay'],
             'loss': topo['netem'][1][2].split()[1]}

    parsed_topos.append([path1, path2])

# Run tests where client sends to wlan first
# Topologies are mirrored so we do not need to vary initial path
for topo in parsed_topos:
    #runTests(protocol = protocol, simulation_class =
    #simulation_class, wlan = topo[0], mbb = topo[1], times = 3)
    path1 = {'id': 'wlan', 'init_path': 1, 'bandwidth': 150, 'delay': 5,
             'queuing delay': 0.09, 'loss': '0%'}
    path2 = {'id': 'mbb', 'init_path': 0, 'bandwidth': 5, 'delay': 500,
             'queuing delay': 0.09, 'loss': '0%'}
```python
runTests(protocol = protocol, simulation_class = simulation_class
            , wlan = path1, mbb = path2, times = 3)

break

cls = ''
proto = ''
	ry:
    opts, args = getopt.getopt(sys.argv[1:], 'hc:p:', ['class=', "protocol="])nn
except getopt.GetoptError:
    print(USAGE)
    sys.exit(2)

for opt, arg in opts:
    if opt in ('-c', '-- class '):
        cls = arg
    elif opt in ('-p', '-- protocol '):
        proto = arg
    elif opt == '-h':
        print(USAGE)
        sys.exit()

if cls == '' or proto == '':
    print(USAGE)
    sys.exit(2)

runTestSuite(cls, proto)
```
D Replay Program

This appendix includes the source code for the replay program used to playback web traffic from dependency graphs.

Listing D.1: Source code for the replay program.

```go
package main

import (
    "crypto/tls"
    "flag"
    "fmt"
    "github.com/buger/jsonparser"
    "github.com/headzoo/surf"
    "github.com/headzoo/surf/browser"
    "github.com/lucas-clemente/quic-go/h2quic"
    "io/ioutil"
    "strconv"
    "sync"
    "time"
    "quic"
    "github.com/lucas-clemente/quic-go"
)

type Download struct {
    Id string
    Type string
    Started bool
    Complete bool
    StartTime time.Time
    CompleteTime time.Time
}
```
Lock sync.Mutex

type Comp struct {
    Id string
    Type string
    Time string
    Started bool
    Complete bool
    StartTime time.Time
    CompleteTime time.Time
    Lock sync.Mutex
}

type Obj struct {
    Id string
    Host string
    Path string
    When_comp_start string
    Comps []*Comp
    Download *Download
    Dependers []*Dep
    Root bool
}

type Dep struct {
    Id string
    A1 string
    A2 string
    Time string
}

var (
host = "https://10.0.0.2:6121"

objs [*]Obj

count [*]byte

verbose * bool

multipath * bool

cache * bool

bindAddr * string

pathScheduler * string

streamScheduler * string

bow * browser.Browser

func parse(graphPath string) {
    data, err := ioutil.ReadFile(graphPath)
    if err != nil {
        panic(err)
    }

    start_activity, _, _, err := jsonparser.Get(data, "start_activity")

    // Parse objects
    jsonparser.ArrayEach(data, func(value [*]byte, dataType jsonparser.ValueType, offset int, err error) {
        obj := new(Obj)

        id, _, _, err := jsonparser.Get(value, "id")
        if err != nil {
            panic(err)
        }

        host, _, _, err := jsonparser.Get(value, "host")
        if err != nil {
            panic(err)
        }
    })
pth, _, _, err := jsonparser.Get(value, "path")
if err != nil {
    panic(err)
}
when_comp_start, _, _, err := jsonparser.Get(value, "when_comp_start")
if err != nil {
    panic(err)
}
obj.Id = string(id)
obj.Host = string(host)
obj.Path = string(pth)
obj.When_comp_start = string(when_comp_start)

// Parse computations
jsonparser.ArrayEach(value, func(value []byte, dataType jsonparser.ValueType, offset int, err error) {
    comp := new(Comp)
    comp_id, _, _, err := jsonparser.Get(value, "id")
    if err != nil {
        panic(err)
    }
    comp_type, _, _, err := jsonparser.Get(value, "type")
    if err != nil {
        panic(err)
    }
    comp_time, _, _, err := jsonparser.Get(value, "time")
    if err != nil {
        panic(err)
    }
})
comp.Id = string(comp_id)
comp.Type = string(comp_type)
comp.Time = string(comp_time)

// XXX: To disable sleep times
// comp.Time = "0"

obj.Comps = append(obj.Comps, comp), "comps")

// Parse download
download := new(Download)

download_id, _, _, err := jsonparser.Get(value, "download", "id")
if err != nil {
    panic(err)
}
download_type, _, _, err := jsonparser.Get(value, "download", "type")
if err != nil {
    panic(err)
}
download.Id = string(download_id)
download.Type = string(download_type)
obj.Download = download

if string(start_activity) == download.Id {
    obj.Root = true
}

objs = append(objs, obj)
}, "objs")
// Parse dependencies

var deps []*Dep

jsonparsing.ArrayEach(data, func(value []byte, dataType jsonparser.ValueType, offset int, err error) {
    dep := new(Dep)

    id, _, _, err := jsonparser.Get(value, "id")
    if err != nil {
        panic(err)
    }

    a1, _, _, err := jsonparser.Get(value, "a1")
    if err != nil {
        panic(err)
    }

    a2, _, _, err := jsonparser.Get(value, "a2")
    if err != nil {
        panic(err)
    }

    time, _, _, err := jsonparser.Get(value, "time")
    if err != nil {
        panic(err)
    }

    dep.Id = string(id)
    dep.A1 = string(a1)
    dep.A2 = string(a2)
    dep.Time = string(time)

    deps = append(deps, dep)
}, "deps")

// Add dependencies to objects
for _, obj := range objs {
    for _, dep := range deps {
        if dep.A1 == obj.Download.Id {
            obj.Dependers = append(obj.Dependers, dep)
        } else {
            for _, comp := range obj.Comps {
                if dep.A1 == comp.Id {
                    obj.Dependers = append(obj.Dependers, dep)
                }
            }
        }
    }
}

func checkDependedActivities(id string) bool {
    for _, obj := range objs {
        for _, dep := range obj.Dependers {
            if id == dep.A2 {
                t, err := strconv.ParseFloat(dep.Time, 64)
                if err != nil {
                    panic(err)
                }
                // Handle download activity
                if obj.Download.Id == dep.A1 {
                    obj.Download.Lock.Lock()
                    if t < 0 && !obj.Download.Complete {
                        // Must wait for the entire download to complete
                        obj.Download.Lock.Unlock()
                        return false
                    }
                    if t > 0 && !obj.Download.Started {
                        // Must wait for the entire download to complete
                        obj.Download.Lock.Unlock()
                        return false
                    }
                }
            }
        }
    }
return true
}
// Download has not started yet
obj.Download.Lock.Unlock()
return false

if t > 0 {
    elapsed := time.Since(obj.Download.StartTime)
    waitTime, err := time.ParseDuration(dep.Time + "ms")
    if err != nil {
        obj.Download.Lock.Unlock()
        panic(err)
    }
    if elapsed < waitTime {
        // Still cannot start
        obj.Download.Lock.Unlock()
        return false
    }
}
obj.Download.Lock.Unlock()

// Handle comp activity
for _, comp := range obj.Comps {
    if comp.Id == dep.A1 {
        comp.Lock.Lock()
        if t < 0 && !comp.Complete {
            // Must wait for computation to complete
            comp.Lock.Unlock()
            return false
        }
        if t > 0 && !comp.Started {
            // Computation has not started yet
            comp.Lock.Unlock()
        }
    }
}
return false
}
if t > 0 {
  elapsed := time.Since(comp.StartTime)
  waitTime, err := time.ParseDuration(dep.Time + "ms")
  if err != nil {
    comp.Lock.Unlock()
    panic(err)
  }
  if elapsed < waitTime {
    // Still cannot start
    comp.Lock.Unlock()
    return false
  }
  comp.Lock.Unlock()
}
return true

func compute(obj *Obj, comp *Comp) {
  comp.Lock.Lock()
  if comp.Started {
    comp.Lock.Unlock()
    return
  }
  comp.Lock.Unlock()}
var wg sync.WaitGroup
wg.Add(2)
go func(comp *Comp) {

defer wg.Done()
for {
    // Check whether all depended activities are done
    ok := checkDependedActivities(comp.Id)
    if ok /* Wait for computation */ {
        var sleepTime time.Duration
        if comp.Time != "0" {
            st, err := time.ParseDuration(comp.Time + "ms")
            if err != nil {
                panic(err)
            }
            sleepTime = st
        }
        comp.Lock.Lock()
        comp.Started = true
        comp.StartTime = time.Now()
        comp.Lock.Unlock()
        start := time.Now()
        for {
            elapsed := time.Since(start)
            if elapsed >= sleepTime {
                break
            }
        }
        comp.Lock.Lock()
        comp.Complete = true
    }
}
}
comp.CompleteTime = time.Now()
comp.Lock.Unlock()
    break
}
)(comp)

   go func (comp *Comp) {
       defer wg.Done()
       processDependers(obj, comp.Id)
   }(comp)

   wg.Wait()
 }

func processDependers(obj *Obj, id string) {
   var wg sync.WaitGroup
   for _, dep := range obj.Dependers {
       if id == dep.A1 {
           for _, o := range objs {
               if o.Download.Id == dep.A2 {
                   wg.Add(1)
                   go func (o *Obj) {
                       defer wg.Done()
                       for {
                           ok := checkDependedActivities(o.Download.Id)
                           if ok {
                               get(o)
                               break
                           }
                       }
                   }(o)
               }
           }
       }
   }
}
for _, comp := range o.Comps {
    if comp.Id == dep.A2 {
        wg.Add(1)
        go func(o *Obj) {
            defer wg.Done()
            ok := checkDependedActivities(o.Download.Id)
            if ok {
                compute(o, comp)
            }
            }(o)
        }
    }
}
wg.Wait()
err := bow.Head(pth)
if err != nil {
    panic(err)
}
mayComp = true
}(host + obj.Path)

wg.Add(1)
go func(pth string) {
    defer wg.Done()
    obj.Download.Lock.Lock()
    obj.Download.Started = true
    obj.Download.StartTime = time.Now()
    obj.Download.Lock.Unlock()

    if *verbose {
        fmt.Println(time.Now().Format(time.StampMilli), "GET", pth)
    }
    err := bow.Open(pth)
    if err != nil {
        panic(err)
    }

    obj.Download.Lock.Lock()
    obj.Download.Complete = true
    obj.Download.CompleteTime = time.Now()
    obj.Download.Lock.Unlock()
}(host + obj.Path)

// Wait for download to complete or first data
for {
    if obj.When_comp_start == "1" && mayComp {
break

obj.Download.Lock.Lock()

if obj.Download.Complete {
    obj.Download.Lock.Unlock()
    break
}

obj.Download.Lock.Unlock()

// Handle computations
if len(obj.Comps) > 0 {
    if checkDependedActivities(obj.Comps[0].Id) {
        if !obj.Comps[0].Started {
            compute(obj, obj.Comps[0])
        }
    }
}

processDependers(obj, obj.Download.Id)
wg.Wait()

func browse() error {
    bow = surf.NewBrowser()

    quicConfig := &quic.Config{
        CreatePaths: *multipath,
        CacheHandshake: *cache,
        BindAddr: *bindAddr,
        PathScheduler: *pathScheduler,
        StreamScheduler: *streamScheduler,
    }
tlsConfig := &tls.Config{
    InsecureSkipVerify: true,
}

roundTripper := &h2quic.RoundTripper{
    QuicConfig: quicConfig,
    TLSClientConfig: tlsConfig,
}

bow.SetTransport(roundTripper)

start := time.Now()
for _, obj := range objs {
    if obj.Root {
        get(obj)
    }
}

onCompletion(start)

return nil

func onCompletion(start time.Time) {
    fmt.Println("Number of objects: ", len(objs))
    var longest time.Time
    for _, obj := range objs {
        if obj.Download.Type == "text/html" || obj.Download.Type == "text/css" {
            for _, comp := range obj.Comps {
                if comp.Type == "text/css" {
                    longest = comp.Time
                }
            }
        }
    }
    fmt.Println("Longest object time: ", longest)
}

for _, comp := range obj.Comps {
    if comp.Type == "text/css" {
        longest = comp.Time
    }
}
if longest.Before(comp.CompleteTime) {
    longest = comp.CompleteTime
}

fmt.Println("Obj", obj.Id, obj.Download.Type, ":", objCompletionTime)

elapsed := time.Since(start).Seconds()
fmt.Println("Page Load Time (s):", elapsed)
elapsed = longest.Sub(start).Seconds()
fmt.Println("Render Time (s):", elapsed)

func main() {
    verbose = flag.Bool("v", false, "verbose")
multipath = flag.Bool("m", false, "multipath")
cache = flag.Bool("c", false, "cache handshake information")
bindAddr = flag.String("b", "0.0.0.0", "bind address")
pathScheduler = flag.String("ps", "LowLatency", "path scheduler")
streamScheduler = flag.String("ss", "RoundRobin", "stream scheduler")
flag.Parse()
graphPath := flag.Args()
if len(graphPath) == 0 {
    fmt.Println("Specify input file")
    return
}

parse(graphPath[0])
err := browse()
if err != nil {
    panic(err)
}