Design and Implementation of a Multipath Extension for QUIC

Master-Arbeit
Tobias Viernickel

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Elektrotechnik
und Informationstechnik
Fachbereich Informatik (Zweitmitglied)

Fachgebiet Multimedia Kommunikation
Prof. Dr.-Ing. Ralf Steinmetz

Design and Implementation of a Multipath Extension for QUIC
Master-Arbeit

Eingereicht von Tobias Viernickel
Tag der Einreichung: 29. November 2017

Gutachter: Prof. Dr.-Ing. Ralf Steinmetz
Betreuer: Alexander Frömmgen

Technische Universität Darmstadt
Fachbereich Elektrotechnik und Informationstechnik
Fachbereich Informatik (Zweitmitglied)

Fachgebiet Multimedia Kommunikation (KOM)
Prof. Dr.-Ing. Ralf Steinmetz

Erklärung zur Abschlussarbeit gemäß §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Tobias Viernickel, die vorliegende Master-Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Master-Arbeit stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Darmstadt, den 29. November 2017

_____

Tobias Viernickel

Contents

List of Figures

List of Tables

Abstract

QUIC is an encrypted, multiplexed, low-latency transport protocol, providing reliable in-order data transfer. Beside these features, QUIC is missing one essential feature, the support of multipathing. Multipathing is a technique to send data, redundant or split over multiple paths, utilizing all available network resources. Recent protocols, such as MPTCP and SCTP, that support the use of multiple paths, tend to appear increasingly. However these protocols have a number of difficulties and disadvantages. SCTP struggles with protocol adoption while MPTCP suffers from packet dependencies between subflows. A multipath-enabled QUIC could overcome these difficulties and leverage multiple network interfaces, such as WiFi and LTE on todays mobile devices, and become the universal stream transport protocol in today's Internet.

We present our design of a multipath extension for QUIC, denoted as MPQUIC. Our design features decoupled packet number spaces and congestion controllers for each subflow. Reliability mechanisms, flow control and encryption is performed on connection level. For subflow establishment we introduce Announcement Packets which are implicitly confirmed by transmitting data, allowing subflow usage after a half RTT. QUIC's native support for multiple streams reduces head-of-line blocking - a well known MPTCP issue - and enables fine-grained stream to subflow scheduling. The implemented prototype of MPQUIC benefits scheduling optimization by providing exchangeable stream and subflow schedulers.

Further we show evaluations of MPQUIC in comparison with traditional QUIC, TCP and MPTCP. The speedup of MPQUIC and the download time behavior is analyzed in homogeneous and heterogeneous networks. The results show that MPQUIC increases throughput in comparison with traditional QUIC, TCP and the currently de facto multipath transport protocol MPTCP. Moreover, we highlight that MPQUIC's conceptual advantages over MPTCP efficiently reduce head-of-line blocking in heterogeneous environments.

1 Introduction

Simultaneously with the era of smartphones, the number of hosts that use multiple network interfaces rises. Mainly smartphones, but also laptops and data centers, provide the hardware to use multiple paths for data transmission. A protocol supporting multipath operations has the ability to provide higher throughput, leading to a more efficient usage of the network resources. In combination with improved resilience to network failures, multipathing enhances the user experience for users with multiple network interfaces [16]. Knowing about the rise of multiple network interface devices, a protocol that makes proper use of the multipath potential is very appreciated in today's Internet.
In the past there were already many attempts to develop protocols that provide mechanisms to establish and utilize multiple paths. A substantial progress was made with *the extension of TCP to support multipath operations with multiple addresses (MPTCP)* [17]. It does exactly what it is supposed to: Enable TCP the use of multiple paths between two endpoints. Before MPTCP the TCP/IP communication was limited to a single path per connection. However the support of multipath operations for TCP has a number of difficulties and disadvantages which are attributable to its TCP roots. In order to remain compatible with TCP and to reduce middlebox interference, MPTCP uses a complex set of TCP options, including a second sequence number space and additional checksums. Moreover MPTCP provides only a single stream to applications on top, forcing them to send all data on this stream and thus adding dependencies between all the data. Furthermore TCP and thus MPTCP guarantees strict in-order delivery which increases head-of-line blocking.
Recent innovations, such as MMPTCP [31] and MPUDP [38], perform well on their specified domain e.g., data centers and virtual private networks respectively, but are not well equipped for general purpose. Many of existing general purpose protocols, such as MPTCP [4] and SCTP [20], are implemented in the kernel. Thus operation system support is required. This also means that prior deployment exhaustive evaluations of the design have to be done. When upgrading or fixing a protocol, again there is a long period of testing and evaluation required until the upgrade is shipped out with the operating system (OS). Hence the evolution process of a protocol, its adoption and innovations after deployment is slowed down. In particular mobile devices are known to have long OS update intervals, which makes protocol upgrade periods even longer.
Another important aspect, which is based on the architecture of today's Internet, is the interference with middleboxes. Deployability of a novel protocol is a huge issue. Middleboxes frequently modify packets and sometimes even make protocols useless [9]. To enable a new protocol deployment on the public Internet, the existing Internet layering must be used. Therefore a new transport protocol is ideally on top of an existing transport protocol [50]. While knowing about all the problems associated with existing multipath transport protocols and being aware of the benefits of a proper multipath solution, we propose a multipath enabled QUIC.
The *Quick UDP Internet Connections* (QUIC) protocol is a novel transport protocol originally developed by Google and is now being standardized by th IETF[1]. QUIC is inspired by and developed with the experience of TCP, SCTP and other transport protocols. Using this experience and being aware of other protocol's limitations benefits the design of QUIC. The importance of deployability encouraged the QUIC designers to utilize UDP with a bunch of features on top of it. Reliability, in-order delivery, congestion and flow control are only a few notable features of QUIC, already well known from other transport protocols. These features, which are until now associated with TCP, appear via QUIC on the application layer on top of the very primitive UDP. Additional security features, multiplexing support and a streaming concept as in SPDY and HTTP/2 makes QUIC effectively like TCP, TLS, SPDY and HTTP/2. Further key strengths, according to the designers of QUIC, are reduced time for connection establishment, advanced congestion control with rich signaling and flow control on stream and connection level [28].

[1] The Internet Engineering Task Force URL *https://www.ietf.org/*

Beside the broad set of features, QUIC was initially conceived to support multipathing. Unfortunately the multipathing approach for QUIC was neither described nor fully implemented, which further propagates the demand for a QUIC multipath extension. QUIC's implementation in the user space on top of UDP, featuring stream multiplexing and full authentication has the most suitable design for a multipath protocol. The ability to multiplex multiple streams over a single connection decreases packet dependencies and decreases head-of-line blocking in heterogeneous networks. Knowing that MPTCP suffers from the nature design of TCP and adopts some of TCP's disadvantages literally forces us to develop MPQUIC, the multipath extension of QUIC.

1.1 Problem Statement

QUIC is a universal transport protocol with many notable features. However QUIC does not support the use of multiple paths. A design and implementation of multipath QUIC (MPQUIC) needs to be investigated. Different possible design approaches must be examined. Based on the implementation, MPQUIC's ability to utilize multiple paths must be evaluated. This includes to analyze the achievable throughput and a comparison with traditional QUIC. Moreover the behavior of MPQUIC for different networks characteristics and traffic patterns needs to be compared with a state-of-the-art multipath protocol.

1.2 Contribution

The contribution of this document is the design, implementation and evaluation of MPQUIC. The objective of the MPQUIC design is to serve as a universal multipath transport protocol. We analyze different possible approaches (e.g., for the subflow establishment) and select the most suitable one for general purpose. The provided implementation is a MPQUIC prototype, that is a multipath extension for QUIC. This prototype enables easy scheduling modifications through an exchangeable stream to subflow scheduler. We proof the concept of MPQUIC by constructing simple test setups, that show the throughput improvement when using MPQUIC compared to QUIC. Additionally we analyze the aspect that theoretical maximum speedups are hard to achieve. The evaluations of MPQUIC provide a comparison against QUIC, MPTCP and TCP. Homogeneous networks as well as heterogeneous networks are simulated using Mininet and the download time when loading files of different sizes is compared.

1.3 Outline

In Chapter 2, we give an inside in background knowledge and protocols that are related to QUIC. Chapter 3 provides information about the QUIC protocol itself. The main contribution of this document is presented in Chapter 4, the design of a multipath QUIC extension. The implementation of MPQUIC's prototype is described in Chapter 5. Before we evaluate and compare MPQUIC with the state-of-the-art protocols MPTCP, TCP and QUIC in Chapter 7, a brief analysis of the prototype is provided in Chapter 6. We complete this document with a conclusion and envision the future work with MPQUIC.

2 Background: Fundamentals and Related Work

The QUIC protocol is not designed and developed from scratch. Instead it combines a set of features that have been proofed to be valuable in other protocols. This chapter gives an inside in background knowledge and protocols that are related to QUIC. First an introduction into fundamental elements of the transport and application layer (e.g., congestion control, flow control, stream-multiplexing) is given. Based on this features of the transport layer, several transport protocols are described. Beginning with the most relevant transport protocols, UDP [46] and TCP [47] are briefly described. UDP is one of the two most used transport protocols of todays Internet. Various other protocols such as CUSP and QUIC run on top of UDP. The other primary transport protocol TCP is responsible for the most of todays Internet traffic. When reliability or in-order delivery is required, TCP is most often used. Beside TCP and UDP the less used but very related protocols SCTP [56], SST [18], CUSP [64], DCCP [32] and MPTCP [17] are briefly introduced. Those protocols are less common than TCP and UDP but have a number of appreciable features which are also present QUIC.
When looking at the application layer a general understanding of HTTP [5, 13, 3] and SPDY [4] is useful for working with QUIC. While some essential HTTP concepts are easily applicable via QUIC and both run interleaved, SPDY can be seen as a predecessor of QUIC and HTTP/2.
Finally for designing a multipathing protocol, it is important to have some background knowledge about schedulers. Therefore a brief introduction into schedulers and the Lowest-RTT-First scheduler is given at the end of this chapter.

2.1 Elements of the Transport and Application Layer

Common techniques for transport and application protocols that support more than just simple packet delivery are congestion control and flow control for an efficient use of the network resources and packet loss recovery for reliability. Other concepts like multiplexing and multihoming that appear in the transport and application layer with increasing frequency are used by many novel protocols as well. To understand the key features of the different protocols, introduced later in this chapter, a description of the most important protocol features, concepts and phenomena is given.

Congestion Control

Participants of a network usually neither have knowledge about the capacity of the network nor how many hosts are using a certain path at the same time. If every sender sends with an arbitrary data rate, the network or a part of it will quite likely being congested after a short time. A congested network causes packets to drop, delayed queuing or blocking of new connections. Actually the network throughput may decrease.
To prevent congestion, the network and the transport layer uses congestion controllers. While the network layer can detect a congestion quite early, the transport layer is responsible for sending packets more slowly. In the following only the transport congestion controller is treated. The design of the congestion control is one of the most important transport elements. Both fairness among other protocols and high utilization of the available bandwidth highly depend on the congestion control [30]. Hence the goal of a congestion controller is not only to avoid congestion but also to find a good allocation of bandwidth to the transport entities that are using the network [58].
A congestion control used in TCP is based on a network congestion-avoidance algorithm [55]. Basically a congestion window (CWND), which is held by a transport entity, is increased at a given interval (i.e., every RTT) by a certain value when no congestion was detected. The initial value for the congestion window is protocol and implementation dependent. Large-scale measurements of the initial TCP window

size reveal that network dependent initial congestion windows are used in today's Internet. In case of HTTP based services, a initial window of 10 packets is most often used [51]. The value for increasing the CWND depends on the current phase of the congestion controller. Slow start, congestion avoidance and fast retransmit are examples for such congestion phases [55]. In this document we will not go deeper into the different values and the conditions for the phases. When reaching the limit of the CWND a sender is not allowed to send data until the CWND is increased by the congestion controller.
The optimal solution for both, single and multipath congestion controllers is under active research and the field is still changing. The many recent congestion designs and evaluations proposed ([6, 37, 48, 65] to name just a few) indicate that creating a efficient and fair congestion control has a large design space. A popular congestion controller, which is currently the default TCP algorithm in Linux, is CUBIC [24]. This congestion controller is also reimplemented for QUIC and basically modifies the growth function of the congestion window.

Flow Control

At the transport and application layer, a flow controller protects the destination buffers from getting overran by a faster sender. In case a sender transmits packets too fast and the receiver can't process all the incoming data packets, buffers might overrun and packets can get dropped. The objective of a flow controller is to prevent buffer overflow at the receiver side while making the most efficient use of network resources. It does so by regulating the senders rate of data transmission. Beside the transport and application layer, flow control can be located at hop, entry-to-exit or network access level[20].
A common flow control mechanism at the transport and application layer is the *sliding window* in combination with a *credit-based* approach. The window in this approach is held by the sender and contains a certain byte size. The sender is allowed to send as much data as the window's byte size allows. If the window limit is reached, transmission must be stopped until a larger window is received. To increase the window size a receiver has to send window updates, the credit. These credits inform the sender about additional window size that may be used for data transmission.

Packet Loss Recovery

Reliability in transport protocols is usually combined with notifications that data has been received by the contracting endpoint. After receiving a notification it is assured that the other endpoint received the data. However reliable delivery is not the norm. Networks can have multiple reasons for losing packets. Thats where packet loss recovery operates to ensure reliable transport of data. When a packet is detected as lost by the sender (i.e., no notification received) the payload of the packet is queued again for retransmission. The recovery of lost packets can be handled by the transport layer (e.g., TCP) or the application layer (e.g., QUIC).
Note that the detailed design of a lost packet detection and packet retransmission is the responsibility of a transport protocol. Also several protocols do not guarantee any level of loss recovery.

Multiplexing and Multipathing

The sharing of several conversations over a single connection is called multiplexing [60]. At the transport or application layer stream multiplexing is an increasingly deployed technique (e.g., HTTP/2, SCTP and QUIC use stream multiplexing). Stream multiplexing is a technique where an application may uses several streams on a single connection to transmit data (e.g., multiple request/responses) interleaved and concurrent. The interleaving data parts of all streams can be separated at receiver side and the per stream data can be reassembled. A notable benefit of stream multiplexing, in contrast to a single stream, is the counteract to head-of-line blocking. If data parts of a particular stream is lost, the other streams

are theoretically not blocked and can continue to be processed. However whether the receiver is blocked or not depends on the individual protocol.
Another application area of multiplexing is the usage of multiple paths. A host owning multiple paths can use inverse multiplexing to distribute the traffic among these paths [60]. This special case is termed multipathing. The receiving protocol reassembles the data received from all paths. After reassembling there is theoretically no difference between single or multipathing.

Multihoming

Connecting a host to more than one external link of potential multiple Internet service providers (ISPs) is called multihoming. The goal of multihoming is to have redundant Internet connections for improved reliability or increased performance. When being connected to more than one ISP, link-level and provider-level fault tolerance is increased [1]. If one path fails another path over another ISP can be used.
Common operators of multihoming are data centers, large enterprises and content providers that depend on the Internet. They require a high level of reliability to operate their businesses [1]. Another common example for a device that could support multihoming is a mobile phone which can be connected to a WiFi and a cellular data network. Each of the network interfaces has their own address.

Head-of-line Blocking

A phenomena termed *Head-of-Line Blocking* (HoLB) can occur when a protocol offers ordered or partial-ordered delivery. HoLB occurs when a packet gets lost and other received data that depends on the lost packet can not be processed. These data is delayed until the lost packet is successful retransmitted. The lost packet is metaphorical the head of the line an it blocks or delays all dependent packets [54].
Head-of-line blocking can occur at different intensities in every protocol that guarantees full or partial ordered delivery. Since HoLB is a consequence of ordered delivery the goal must be to minimize dependencies of transmitted data. Less dependencies allows receiving applications to continue processing received data even though precedent packets are lost. Scharf and Kiesel [54] stated that many signaling protocols in IP networks do not require strict in-order delivery but still use TCP, that as we will see later always guarantees full in-order delivery. The fully ordered data delivery of TCP may unnecessarily increase the end-to-end delays because of HoLB.
On the contrary the impact of HoLB is ambiguous according to the work of Grinnem et al. [23]. By using unordered delivery on a constant SCTP flow the authors could only decrease the average message transmission delay by 0% up to 18% compared to ordered delivery. The impact of HoLB was statistically insignificant in several tests.
Especially because the results of studies that examined HoLB are ambiguous it is important to consider this phenomena when constructing a new protocol or any extensions to it.

2.2 Transport Layer Protocols

The transport layer protocols provide data transport between two processes on potentially different machines and additionally a certain level of reliability [62]. The transport layer and the belonging protocols also provide an abstraction of the underlaying physical network to higher layer protocols.
Higher layer protocols such as HTTP/2 or QUIC have ongoing communication via the network. Since these high layer protocols overlook the underlaying physical network the abstraction provided by the transport layer is necessary in order to use the network.
The main transport protocols of todays Internet are UDP and TCP [63]. While the former is connectionless and does not much more than sending packets, the latter is connection-oriented and provides plenty

of additional features [63]. Other less common protocols have features like streaming or multi-path support which can also be found in QUIC.

2.2.1 UDP - User Datagram Protocol

The simplest transport layer protocol is the *User Datagram Protocol* UDP. UDP is specified by Postel [46] in RFC 768. By using UDP an application can send datagrams over the Internet Protocol IP without a connection establishment e.g., there is no handshake. UDP is designed to provide transport of messages with a minimum of protocol mechanisms. In order to fulfill this design, UDP is kept very simple and neither delivery nor duplicate protection are guaranteed.
The UDP header consists of a source port, a destination port, the length of the user datagram and a checksum. While the source port is optional and can be used for addressing the reply, the destination port in combination with the ip address of the IP header build a particular Internet destination address [46]. A process attached to the destination port receives the payload of the user datagram. The length field contains the total length of this header and the data transmitted by this datagram. The optional checksum is used by the receiver to assure the datagram was transmitted correctly. It comprises the header, the payload and a conceptual IP pseudo header [63].
The User Datagram Protocol is used by the application layer protocol QUIC as transport protocol. As in more detailed described in chapter 3, QUIC is connection-oriented and provides reliability, in order delivery with congestion and flow control. All those QUIC properties are traditional features and elements of the transport layer [62]. This fact makes it worth to state that traditional UDP does not provide any of the recent mentioned features.

2.2.2 TCP - Transmission Control Protocol

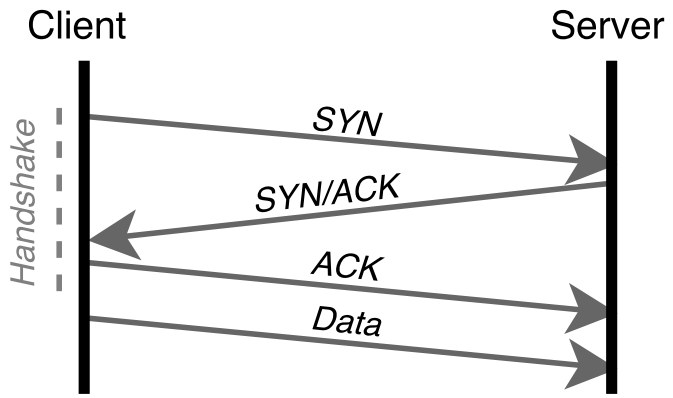


Figure 2.1.: Timeline of a successful TCP connection establishment

The *Transmission Control Protocol* TCP is the most used protocol when reliable in-order delivery is required [61]. TCP is designed to provide connection-oriented and reliable transport between two machines over an unreliable internetwork. It was first defined 1981 by Postel [47] in RFC 793 and from then on many extensions and improvements have been made. RFC 793 is the fundamental specification of TCP. RFC 4614 [10] serves as a roadmap for TCP and its evolution. Together with UDP, TCP forms a complement which provides the most services of todays Internet. Note that neither UDP nor TCP provide a secure connection. For data encryption an additional transport layer security (TLS) protocol can be used.
On every machine that supports TCP runs a TCP transport entity. This entity is responsible for receiving

the data, processing the data and handing the data over to the destination application. Applications that use TCP over UDP usually require good performance on one side but also good reliability and ordered delivery on the other side. Although some applications don't need a strict ordered delivery as TCP provides and other protocols might suit better for their use cases, TCP is often the default choice for reliable transport in the Internet [54].
For the communication with other applications, TCP connects them with a TCP stream. During connection establishment, depicted in Figure 2.1, TCP performs a 3-way-handshake performed. First the initiator sends a SYNchronize to a peer it wishes to build up a connection. If the receiving peer accepts the SYN, it answers with a SYN in combination with a ACKnowledgement. The handshake is completed and the connection established after a final ACK sent by the initiator. An application can then pass data to the TCP entity. Next the data is sliced into smaller pieces so they have the required size for an Ethernet frame. Each of this slices is send by TCP as a separate IP datagram. The receiving machine in turn hands the datagram to the TCP entity which reconstructs the original data out of the received slices. In case a datagram was received out of order the TCP entity is also responsible for restoring the correct byte order[61].
For reliability and performance reasons TCP uses a congestion controller, a flow controller, packet loss recovery but also suffers from head-of-line blocking since data is delivered in fully order.

2.2.3 Further related Protocols

In the following the transport layer protocols SCTP, SST, DCCP, CUSP and MPTCP, which are less common than UDP and TCP, are introduced. Even thought they are not widely spread, they provide features which are adopted into QUIC. Concepts like multi-homing, multi-streaming and multipathing already have a rich history and are no new invention for QUIC.

SCTP

The *Stream Control Transmission Protocol* (SCTP) [56] is a reliable but connectionless transport protocol on top of the IP network service. It is message oriented like UDP but provides features like reliability, in-order delivery and congestion control known from TCP. The biggest difference between the popular TCP and SCTP is multi-homing. Through the support of multi-homing, SCTP is able to establish even more robust communication associations between two endpoints. These endpoints may have several network interfaces which may lead to different data paths. The presence of multiple paths introduce the need for an adjusted congestion control. SCTP provides the required multi-path congestion control by handling separate congestion control parameters for each path.
As the name implies the SCTP protocol uses a stream concept. It supports sequenced delivery of user messages within multiple streams. The concept of multiple streams allows SCTP to relax the in-order delivery and an application does not need to wait until every potential gap in the transport sequence number is filled. Data can be delivered to the upper-layer protocol if the sequence numbers of a particular stream are in-order or if a certain flag (unordered mode) is set that disables ordered delivery.
Scharf and Kiesel [54] showed that SCTP streams or SCTP unordered mode can avoid head-of-line blocking while the default choice protocol TCP may suffer from significant delays even for moderate packet loss probabilities.

SST

The *Structured Stream Transport* (SST) [18] is a transport abstraction that uses a multiple-stream concept. The goal of this experimental protocol is to address applications that have multiple parallel communication activities such as loading different parts of a web page.

SST also support reliable, best effort delivery, optional end-to-end cryptographic security and dynamic stream prioritization. The latter one allows an application to prioritize its streams relative to each other and adjust the priorities.
Even thought SST was never widely spread, for Ford [18] the structured streamed appeared to be a promising improvement to the common reliable stream abstraction.

DCCP

For large amount of data that need congestion control but no reliability (e.g., video or music streaming) the *Datagram Congestion Control Protocol* (DCCP) [32] was developed. DCCP is a transport protocol on top of the IP network service that transports an unreliable flow of datagrams. In contrast to UDP it has additional congestion control, a reliable handshake for connection establishment and acknowledgements. With these features DCCP provides a controllable tradeoff between delay and reliable in-order delivery.
DCCP is a protocol that has a built in congestion control mechanism. To specify different mechanisms a Congestion Control Identifier (CCID) is used. The most common CCID profiles are TCP-like Congestion Control (CCID 2) [14] and TCP-Friendly Rate Control (CCID 3) [15].
Applications that rather have timeliness in packet delivery than reliability are the target group of DCCP. For this reason Chowdhury et al. [8] performed experiments using CCID 2 and CCID 3 on applications that suffer the tradeoff between delay and in-order delivery. Based on the results the authors stated DCCP with CCID 3 can be a better choice for real time applications than TCP.

CUSP

The *Channel-based Unidirectional Stream Protocol* (CUSP) is a reliable general purpose transport protocol designed for applications that use peer-to-peer networking. The previously mentioned SST and SCTP provided the ideas for developing (CUSP) [64]. For connecting two applications CUSP uses channels. Channels are responsible for the low-level packet management and provide negotiation, congestion control and cryptography. Inside these channels are multiplexed streams. Known from other multi-stream protocols, streams do not block each other. Ordered delivery is only ensured inside each stream.
On the top of UDP, CUSP is easy to deploy but requires a single round-trip handshake unlike UDP. Just as SST, CUSP also supports stream priorities. The stream with the highest priority that is ready always sends first.
The authors claim that CUSP had the potential to replace the already discussed protocols DCCP, SCTP and SST because of its wide range of features from unidirectional streams to encryption [64]. Experiments where they measured the network performance showed that their CUSP prototype can keep up with TCP and outperform SCTP and SST.

MPTCP

With the objective to support the usage of multiple physical paths between two endpoints a set of *TCP extensions for multipath operation with multiple addresses* (MPTCP) [17] were developed. Before MPTCP the TCP/IP communication was limited to a single path per connection. The growing number of devices that have multiple paths (e.g., WiFi and cellular networks used by smartphones) can enable improved resource usage within the network. With MPTCP it is possible to simultaneously use multiple paths between peers. It also offers the same services (e.g., reliability, ordered delivery) to applications as TCP.
MPTCP is settled in the transport layer on top of potential several TCP connections. Each of these TCP connections is created by the MPTCP implementation for every physical path, termed MPTCP subflow. Because they are real TCP connections each subflow has their own subflow sequence number

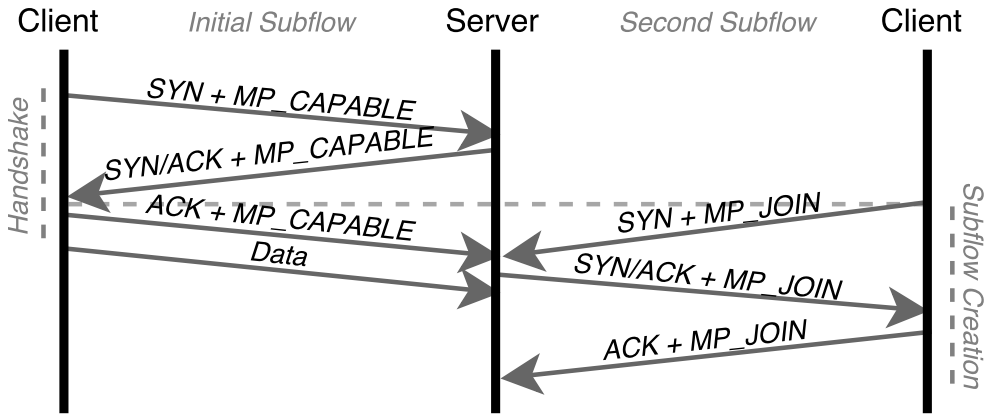


Figure 2.2.: Timeline of a successful MPTCP connection and subflow establishment

(SSN) space. These subflows look like regular TCP connections to the network layer. In contrast for an application a MPTCP connection including all subflows looks like a single TCP connection. To a non-MPTCP-aware application, every MPTCP implementation will behave as a normal TCP implementation. Even the MPTCP connection establishment is similar to the TCP connection establishment.
For signaling the MPTCP operations inside a TCP packet, the optional TCP header fields are used. To inform a receiver about the willingness to open a MPTCP connection, instead of a TCP connection the operation MP_CAPABLE is sent in the first message (SYN) during the connection establishment. If the server supports and agrees on using MPTCP it sends the MP_CAPABLE in the response message (SYN/ACK). The agreement about MPTCP is successful accomplished after the client again sends the MP_CAPABLE in the third message (ACK) of the 3-way Handshake. If any of those MP_CAPABLE is missing (e.g., server does not support MPTCP, middlebox deleted MP_CAPABLE option) a ordinary TCP connection is established. Adding additional subflows uses the same routine as initiating a normal MPTCP connection but the SYN, SYN/ACK, and ACK packets carry the MP_JOIN option instead of MP_CAPABLE. Figure 2.2 shows the timeline of a MPTCP connection establishment and the creation of an additional subflow. Note that similar to TCP, the transmission of the data is not secured.
For in-order delivery among subflows a data sequence number (DSN) is used. With the DSN the stream data as a whole can be reassembled by data sequence mapping components, which provide a mapping between DSN and SSN.
Since MPTCP looks like normal TCP for applications and there is a delivery order between subflows, data can only be passed to the application if there is no gap in the data sequence numbers. As consequence if a packet of any subflow is lost, all received packets of all subflows with a DSN grater than the DSN of the lost packet can not be delivered to the application. This phenomenon is called head-of-line blocking as described in section 2.1.

2.3 Application Layer Protocols

The described transport layer protocols serve as abstraction for an unknown underlaying physical network. This abstraction is used by application layer protocols in order to communicate with applications on other machines. While the protocols below this layer provide transport services the protocols on the application layer can be used by user applications. These applications directly work with user data and use application layer protocols as entry point to the Internet.
Omitting other supportive protocols on the application layer this section will introduce HTTP and SPDY as specific application layer protocols. HTTP and SPDY are closely related to QUIC as we will see in Chapter 3.

2.3.1 HTTP

The *Hypertext Transfer Protocol* HTTP is a widely used application layer protocol and the basis for the World Wide Web [59]. HTTP is a stateless protocol and uses a client/server model. The most known application that uses HTTP is a Web-browser.
HTTP/1.1 which was described by Fielding et al. [13] in RFC 2616 specifies the first official standard of HTTP. Since its publication in year 1999 it serves as a foundation for the growth of the Internet until now. However the demands of todays Internet have changed a lot. Permanently increasing numbers of everyday interactions continue to migrate to the web [21]. As a consequent protocol performance and near-realtime responsiveness is required. Following this demand HTTP/2 was developed and specified in RFC 7540 by Belshe et al. [3].
The goal of HTTP/2 was not to replace HTTP/1.1 rather then to extend it by making it faster, simpler and more robust. Therefore the high-level API of HTTP/1.1 and HTTP/2 are the same. The improvement of the performance is achieved through low level changes [22].
HTTP/2 has a number of mechanisms and concepts (e.g., multiplexed streaming, frames as basic units) that are also implemented in QUIC [25]. These mechanisms of HTTP/2 can be replaced by QUIC's implementation. Since HTTP/2 is specified to be an application-layer protocol running on top of a TCP connection [3], replacing some mechanisms by QUIC, which is running on top of UDP, is not conform with the HTTP/2 specification. Rather the new term HTTP over QUIC (HTTP/QUIC) was introduced in the QUIC community. HTTP/QUIC describes a mapping of HTTP semantics over QUIC and identifies HTTP/2 features that are also implemented by QUIC [40].

HTTP/2

HTTP/2 is used for communication between clients and servers over TCP. For communication HTTP messages are used. There are two types of messages called *Requests* and *Responses*, that both contain a message header and a message body. While the header contains information (e.g., coding, content-type) about the body, the message body is used for transmission of the user data. The data in turn is subdivided into frames, the core transmission unit for user data.
As already seen for several transport-layer protocols (e.g., SCTP, SST) HTTP/2 and HTTP/QUIC uses a multiplexed streaming concept. Multiple streams can be opened or closed independently during a single connection without an additional RTT. Each stream has a stream ID with a predefined purpose. For the usage with QUIC, a individual HTTP/QUIC stream mapping is provided (Section 3.1.5).
HACK: HPACK is the Header Compression for HTTP/2 [45]. For compression, HPACK treats a list of HTTP header fields as an ordered collection and encodes this collection. As HTTP is designed for the use with TCP and TCP guarantees fully-ordered delivery it is justified that HPACK relies on an ordered header list. However, when using HPACK for unordered data transfer, additional head-of-line blocking is induced.

2.3.2 SPDY

The experimental SPDY [4] protocol was developed at Google and intended to reduce web page load latency. Before SPDY was replaced by HTTP/2 it was supported on an increasing number of websites and web browsers. The SPDY trend in year 2012 triggered the development of HTTP/2 whereby the SPDY specification served as starting point [22]. Even thought SPDY is now deprecated it is worth mentioning because it can be considered as predecessor of QUIC.
Key features of SPDY are reliable transport, stream multiplexing and prioritizing of streams. A single connection can be used for multiple request when using SPDY. The base unit for transport are frames. Many mechanisms of SPDY have been adopted to QUIC and parts of the SPDY implementation can still

be found in the QUIC chromium source code[1]. However one of the main differences to QUIC is the rely on a reliable transport protocol for instance TCP. This usage of TCP again leads to the same problem of head-of-line blocking as already discussed in section 3.1.5.

2.4 Schedulers

The availability of more than one network path allows a multipath supporting protocol, such as MPTCP, to utilize additional bandwidth for sending data. In general the available paths are heterogeneous duo to different RTTs and different bottleneck capacities. A multipath protocol may has the objective to minimize the required time for data transmission with respect to the available capacity. In theory the capacities can be seen as renewable resources that is used for performing a certain activity (i.e., transmit a certain amount of data). This sets up a general scheduling problem called resource-constrained project scheduling problem [2].
In general multipath aware protocols implement schedulers that distribute outgoing data on different subflows. However the scheduling decision is not trivial and a wrong scheduling decision can significantly affect download times [44]. The design space for schedulers is very broad. Existing schedulers send on the path with the lowest RTT (Lowest-RTT-First), penalize retransmission (Retransmission and Penalization) or simply rotate the subflows for sending (RoundRobin).

2.4.1 Lowest-RTT-First Scheduler

An experimental evaluation of schedulers for the de facto multipath protocol MPTCP proposes a Lowest-RTT-First (MinRTT) scheduler as general purpose scheduler for MPTCP [44]. The MinRTT scheduler reduces application delay, which improves the user-experience, by scheduling data on the subflow with the lowest RTT. At some point the congestion window of the lowest RTT path might be exhausted. The MinRTT scheduler will then send data on the subflow with the next higher RTT. When all congestion windows are filled, the process becomes blocked until incoming acknowledgements increase the congestion window.

[1] The Chromium Projects: QUIC, a multiplexed stream transport over UDP URL *https://www.chromium.org/quic*

3 Background: QUIC

Quick UDP Internet Connections (QUIC [1]) is the name of the new all-round transport protocol designed to improve transport performance for HTTPS [36]. Formerly developed and promoted at Google, QUIC is now being designed and developed by the QUIC Working Group (quic-wg) of the Internet Engineering Task Force [2].

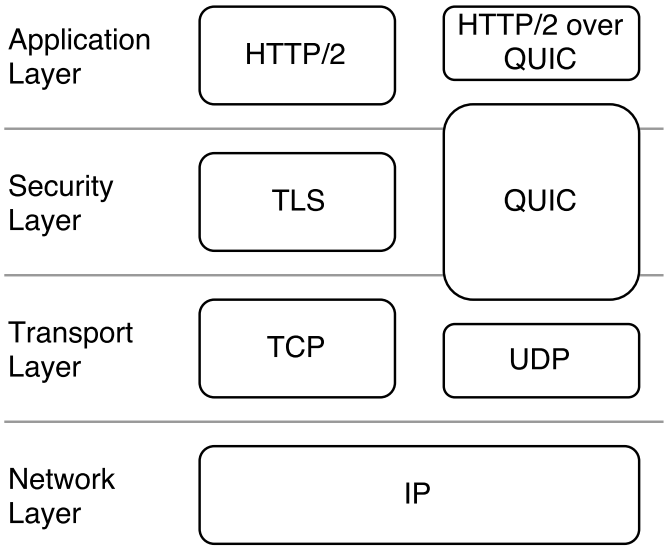


Figure 3.1.: QUIC and TCP in the network stack

Figure 3.1 illustrates QUIC in the network stack. QUIC covers partly the application layer, providing an *Application Programming Interface* (API) that is designed for, but not restricted to HTTP/2 transport. Applications can use multiple streams to send data over QUIC, which in turn sends this data over a single connection. By providing encryption and authentication for most of the packets, equivalent to *Transport Layer Security* (TLS), QUIC completely covers the security layer. Additionally by providing reliable in-order data transfer, congestion and flow control mechanisms, QUIC intersects the transport layer, located on top of UDP.

A distinctive feature of QUIC is a reduced number of RTTs necessary for connection establishment. QUIC does so by sharing both, cryptographic and transport information in one handshake. After a client is aware of the initial keys of a server, it is possible to achieve data transmission on a new connection with 0-RTT delay.

Another strength of QUIC is the richer signaling, appearing at different QUIC mechanisms. Richer signaling means that through design decisions and additional packet information, QUIC naturally benefits its own mechanisms. One example is the delay between receiving an packet and an acknowledgement being sent. This delay is included in each acknowledgement and is used for a precise round-trip time (RTT) calculation. Another aspect is the uniqueness of every single packet number, as we will see later. By having different packet numbers for lost packets and their retransmission, a ambiguity problem, known from TCP, that occurred when distinguishing ACKs is avoided.

Moreover QUIC provides improved deployability, multiplexed streams and reduced head-of-line blocking. This chapter describes characteristics of QUIC, including the packet and frame format and also highlights the key features of QUIC

[1] In this paper, QUIC always refers to Google's version of QUIC[36, 25]

[2] The Internet Engineering Task Force - QUIC Working Group URL *https://datatracker.ietf.org/wg/quic*

Versions of QUIC

The QUIC protocol is under active development leading to continuous protocol changes together with appearing QUIC versions. Google's first QUIC draft appeared in 2012. In midyear 2016 the IETF decided to start with an active protocol standardization process for QUIC . From then development of an IETF branch of QUIC started, resulting in a name ambiguity between Google QUIC (denoted as gQUIC) and IETF QUIC (denoted as ietfQUIC). As the branches of QUIC evolve, there appear to exist even different versions of QUIC for each branch. Recent experiments and the large-scale deployments of QUIC described in [36] are based on gQUIC. While writing this document, the latest public available version is 39.
Note that gQUIC significantly differs from ietfQUIC. The streaming concept, packet formats, frame types and the adaption of TSL 1.3 into QUIC are just a few examples for differences between these branches.

3.1 Packets, Frames and Streams

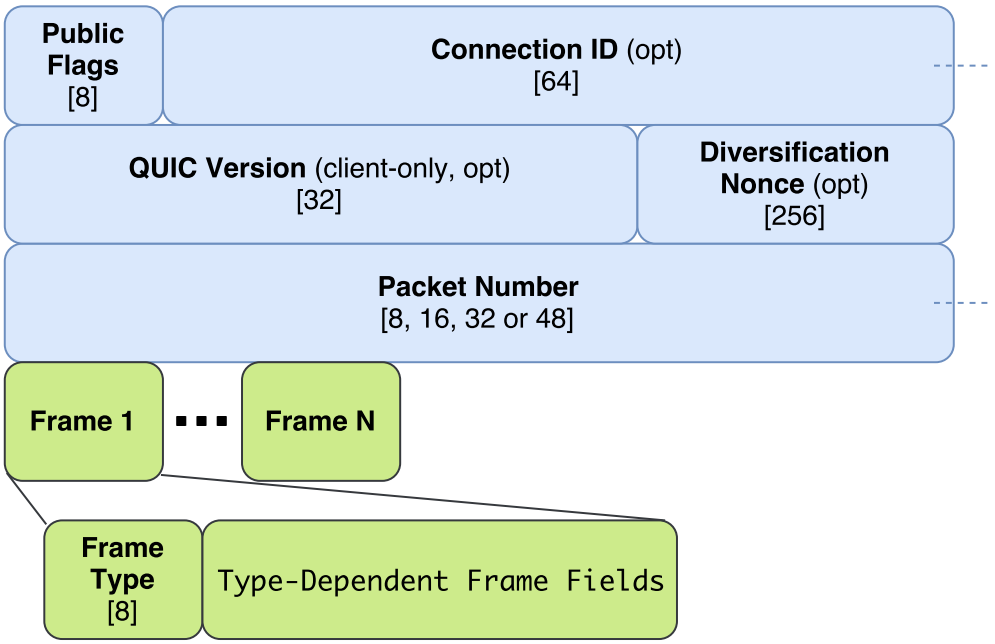


Figure 3.2.: QUIC packet format [36]: Beginning with a public header (blue) followed by an encrypted body (green).

The QUIC protocol uses packets for transmitting data over UDP. The message oriented UDP provides an API to send or receive exactly one datagram per operation (*send* and *receive* respectively). To match this design, QUIC bundles packets of data and hands them over to UDP.
These packets form the payload of a UDP datagram. All QUIC packets begin with a public header, followed by the payload. The payload in turn may include various type-dependent header bytes. In case of the most common packet type, the `Fame Packet`, the payload consists of type-prefixed frames. For transmitting application data, `Stream Frames` are used. These frames contain a `StreamID`, used to realize stream multiplexing. The structure of a QUIC packet is depicted in Figure 3.2.

3.1.1 Packet Types and Formats

In QUIC it is distinguished between `Special Packets` (i.e., `Version Negotiation Packets`, `Reset Packets`) and `Regular Packets` (i.e., `Frame Packets`) (Figure 3.3). The most relevant packet type

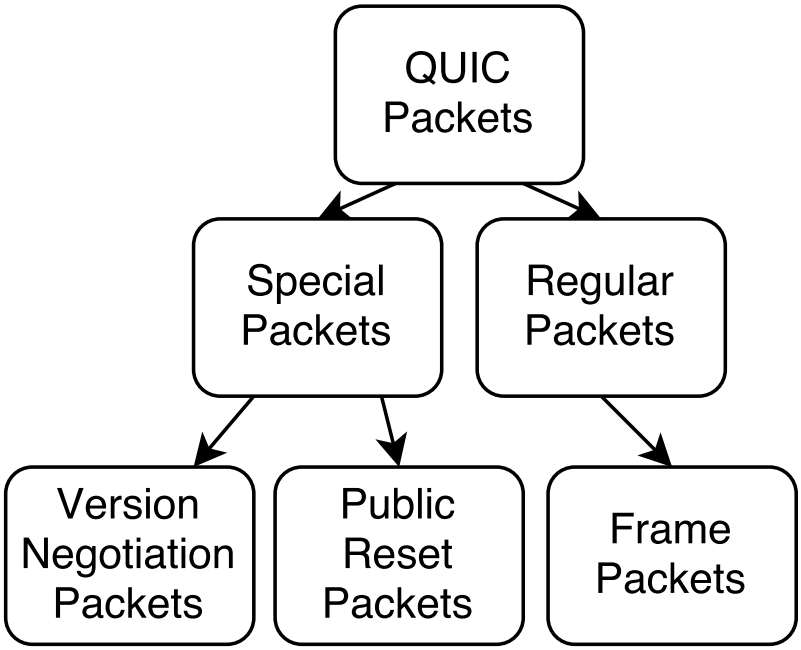


Figure 3.3.: In QUIC there is a distinction between `Version Negotiation Packets`, `Reset Packets` and `Frame Packets`

for data transmission is the `Frame Packet`. It simply contains frames of potentially different types directly after the header.
The predefined overall packet payload size is 1350 bytes. This particular size was determined by Google with wide-scale reachability experiments [36] and will be used until *Maximum Transmission Unit* (MTU) discovery [42], a technique for dynamically discovering the MTU of an arbitrary Internet path, is available in QUIC.
Public Flag: Every QUIC packet starts with individual `Public Flags`, providing the receiver with information about the composition of the packet. This information contains the length of the `ConnectionID`, the length of the `Packet Number`, a `Public Reset Flag`, indication of the presence of the `Diversification Nonce` and a reserved flag (i.e., `Multipath Flag`) for multipath usage. A list of the individual public flags and their meaning is given in Table 3.1.

Table 3.1.: Description of the different `Public Flags` inside the packet QUIC header.

Public Flag	Meaning
0x01	Public Version Flag
0x02	Public Reset Fag
0x04	Presence of Diversification Nonce
0x08	Presence of 8 byte ConnectionID
0x30	(2 bit mask) Indicates length of Packet Number
0x40	Reserved for multipath use
0x80	Not used

ConnectionID: The`ConnectionID` is an unsigned 64 bit random number, used to identify the connection. By using this `ConnectionID` instead of a four tuple (destination and source `IP/Port`) a connection is migrated in case a client roams.
QUIC Version and Version Negotiation A QUIC client proposes a QUIC version to use for the connec-

tion in the first packet of the connection establishment and encodes the rest of the handshake using the proposed version. If a server does not support the version, announced in the QUIC Version, a Version Negotiation Packet is sent by the server. The Version Negotiation Packet carries all of the server's supported versions. The process of version negotiation causes one additional RTT delay for connection establishment. QUIC clients and servers perform version negotiation during connection establishment to avoid unnecessary delays later on. However when the client's optimistically-chosen version is spoken, one RTT at connection establishment is eliminated.

Diversification Nonce A server has the option to send a Diversification Nonce with the header, to add entropy into key generation. Note that the version number and diversification nonce fields can only be present in early packets.

Packet Number: The last segment of the header is the Packet Number. The first packet sent by a client or server starts with 1. Each following packet shall have a larger Packet Number than the previous packet. A more detailed description about Packet Numbers is provided later in this chapter.

3.1.2 Frame Types

The base unit in QUIC for data transmission is a frame, located inside the packet payload. Similar to QUIC's packet concept, there are two main types of frames, i.g., Special Frames and Regular Frames, as well as sub-types. A listing of the frame types in detail is given in Table 3.2. A QUIC packet may contain multiple frames of potentially different streams as well as additional control frames.

Table 3.2.: Different Frame Types in QUIC

Special Frames	Stream, ACK, Congestion_Feedback
Regular Frames	Padding, RST_Stream, Connection_Close, Ping GoAway, Window_Update, Blocked, Stop_Waiting,

Every frame is composed of a type-specific header and type-specific data, whereas the header itself contains the information about the actual type The length of a frame is unlimited theoretically. However frames are not allowed to span across packets.

The most common frames are Stream Fames and Acknowledgement (ACK) Frames. While Stream Fames are used for application data transmission, ACK Frames are necessary for ensuring reliability.

3.1.3 Streams

QUIC enables applications to write data on individual streams. Streams are a independent sequences of data. Here it is possible to make use of multiple streams of a single QUIC connection at the same time. The streams are used concurrently and data can be send interleaved with other streams. Every stream offers in-order delivery while data of different streams may be delivered out of order, thus reducing head-of-line blocking.

Each stream is identified by a unsigned Number, denoted as StreamID, and possesses an own sequence number space (i.e., Offset) to reduce inter-stream dependencies. A detailed description of the stream Offset is given Section 3.2.1. Internally streams are multiplexed using Stream Fames. QUIC uses streams to send both application data (e.g., HTTP messages) as well as QUIC data (e.g., crypto handshake, window updates). To open a stream, a client or a server sends data, marked with an unused StreamID. The appearance of the unused StreamID implicitly triggers the creation of a new stream at the receiving side. This reduces handshake massages and protocol complexity and enables 0-RTT stream establishment. Streams that are not longer needed may be closed by either server or client without closing the whole connection.

3.1.4 Stream Mapping

QUIC streams are opened by either clients or servers. In both cases the stream may be used bidirectional, which allows both peers to send data over a stream. However, as both, client and servers, may open streams, `StreamID` collision must be prevented. For this purpose, a stream mapping is specified.
When opening stream, QUIC specifies that clients initiated streams must have odd `StreamIDs` while server initiated streams obtain even `StreamIDs`. This distinction prevents `StreamID` collisions when opening a new streams. Additionally `StreamIDs` must monotonically increase with every new opened stream.
Note that `StreamID 0` is not a valid Stream ID. `StreamID 1` is reserved for the cryptological handshake. Thus the first stream initiated by the client consumes `StreamID 3`, the second `StreamID 5`. Server streams start with `StreamID 2`, followed by `StreamID 4`. This scheme carries on for every further opened stream.

3.1.5 HTTP/2 Stream Mapping

When using HTTP over QUIC, a specific HTTP/QUIC Stream Mapping is used. This is because HTTP/2 uses HPACK compression for request and response headers. As HPACK requires in-order delivery, the headers must be delivered in-order. However QUIC provides ordered delivery of data only within streams. Thus the HTTP headers must be send over the same stream.
When using HTTP/2, `StreamID 3` is reserved for transmitting HTTP/2 response and request headers. By belonging to the same stream, it is guaranteed that the order of the headers is preserved. The HTTP/2 bodies corresponding to the headers are sent over different streams. When initiated by the client, the first stream must have `StreamID 5`. Server streams start with `StreamID 2`. The remaining mapping is identical to the standard stream mapping.

Head-of-line Blocking in HTTP/2 and QUIC

A notable advantage of HTTP/QUIC over HTTP/2 is the partly elimination of head-of-line blocking. In HTTP/2 over TCP, losing a packet causes every stream to wait for the missing packet. Even if the lost packet has no required stream frames for the following packet, the HTTP/2 entity must wait, as TCP guarantees strict ordered delivery.
On the contrary HTTP/QUIC is on top of UDP. Every transmitted UDP datagram is immediately handed to HTTP/QUIC. If a packet is lost, HTTP/QUIC keeps on processing frames of successive packets that belong to different streams than frames inside the lost packet. This means that only streams that have frames inside a lost packet are blocked.
However, as already mentioned, head-of-line blocking is not fully eliminated. The loss or reordering of a packet containing a header may still cause other independent streams to be blocked, as their headers might depend on the lost header. This is duo to HPACK, the Header Compression for HTTP/2.
A possible solution for this problem is provided by QPACK header compression for HTTP/QUIC [39]. QPACK adjusted HPACK to be more compatible with QUIC by loosen the ordering requirements. However currently QPACK is only a draft and not used by HTTP/QUIC.

3.2 Reliable In-Order Data Transmission

The QUIC protocol applies common mechanisms of the transport layer and guarantees reliable in-order delivery. As UDP does not provide ordered delivery nor reliability, it is QUIC's responsibility to take care of lost or out-of-order packets. Thus every `Regular Packet` in QUIC is marked with a `Packer Number` and every `Stream Frame` carries an `Offset`. On one side, the `Packet Number` in combination with `ACK`

Frames are used for loss detection. The Offset of a Stream Frame, on the other side, ensures that data is delivered to the receiving application in the original order.

3.2.1 Packet Numbers and Stream Offsets

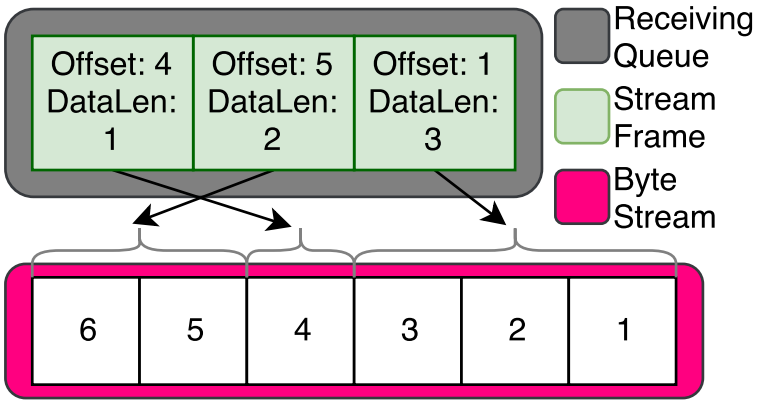


Figure 3.4.: The Offset of a Stream Frame determines its position in the byte stream, which is given to the application on top of QUIC.

QUIC separates the *Packer Number Space* and the *Stream Data Offset*. The main purpose of Packet Numbers in QUIC is the detection of lost packets. Packet Numbers in QUIC are unsigned numbers that appear only once for the entire connection. This is in contrast to other transport protocols, such as TCP or MPTCP, which reuse Packet Numbers in case of retransmission and also order the application data based on this number. By not ordering the data based on the Packet Number, the dependencies between the payloads of the packets are loosen. Avoiding duplicate Packet Numbers prevents ambiguity of fresh and retransmitted packets [36], making duplicate detection trivial [29].
In QUIC, Packet Numbers are used for Regular Packets only. Every sender has to assign a unique Packet Number to every outgoing regular packet, beginning with Packet Number 1. Each subsequent packet shall have a larger Packet Number than that of the previous packet
For stream management, including in-order delivery, every Stream Frame is tagged with an Offset. A visualization of the Offset mechanisms is provided in Figure 3.4. The Offset equals the byte offset, that is the position of the Stream Frame's payload in the byte-stream. Based on this byte offset, the original ordering of the stream data is reconstructed. In case of a gap in the byte-stream (e.g., caused by out-of-order packets), only the corresponding QUIC stream is blocked. Every other stream can continue being processed.

3.2.2 Loss Detection and Retransmission

The detection of lost packets and the retransmission of those is base on ACK Frames, on the loss detection alarm and on a send history
To indicate the receipt of packets as well as to indicate which packets are considered missing, a receiver sends ACK Frames. Every ACK Frame includes aLargest Acked field, representing the peer's largest observed Packet Number. Additionally, every ACK Frame may contain between 1 and 256 ACK Blocks. These blocks represent ranges of acknowledged packets. Between each two ACK blocks is a Gap field. The Gap field indicates missing Packet Numbers.
The multi-modal loss detection alarm is used for time-based loss detection. The various modus determine the duration of the alarm and the action to be performed when the alarm fires. Both, the alarm when firing and a newly obtained Largest Acked trigger the execution of the loss detection in QUIC. Note that

in case the alarm triggered the loss detection, the previous `Largest Acked` is supplied [29]. A packet is considered to be lost, when the `Largest Acked` is larger than the packet number of any sent packet.
The send history stores information for potential retransmission. In case of a packet loss, its payload is reconstructed based on the send history. However, the packet itself is not being retransmitted, rather the payload is rescheduled when constructing new packets for sending.

3.3 Connection Establishment

In QUIC, every connection is established for secure data transport. Therefore additional cryptographically information must be shared between communicating peers. To achieve low-latency connection establishment, QUIC specifies a combination of the cryptographic and transport hand.shake. The cryptographic handshake is executed on the for this purpose reserved reliable stream with (`StreamID 1`) as described in Section 3.1.4. The obtained information is cached by a client for later re-use. Upon establishing subsequent connections to the same peer, a client can re-use this information for a fast connection establishment.
Once a connection is established in QUIC, is can be identified by a `ConnectionID`. A huge advantage of the `ConnectionID` is the possibility to identify a connection even if the IP address or the port of the communication partners change. Reasons for a changing IP address may be roaming peers or NATs. In QUIC address changes trigger connection migration which does not require a separate handshake, enabled by the `ConnectionID`.

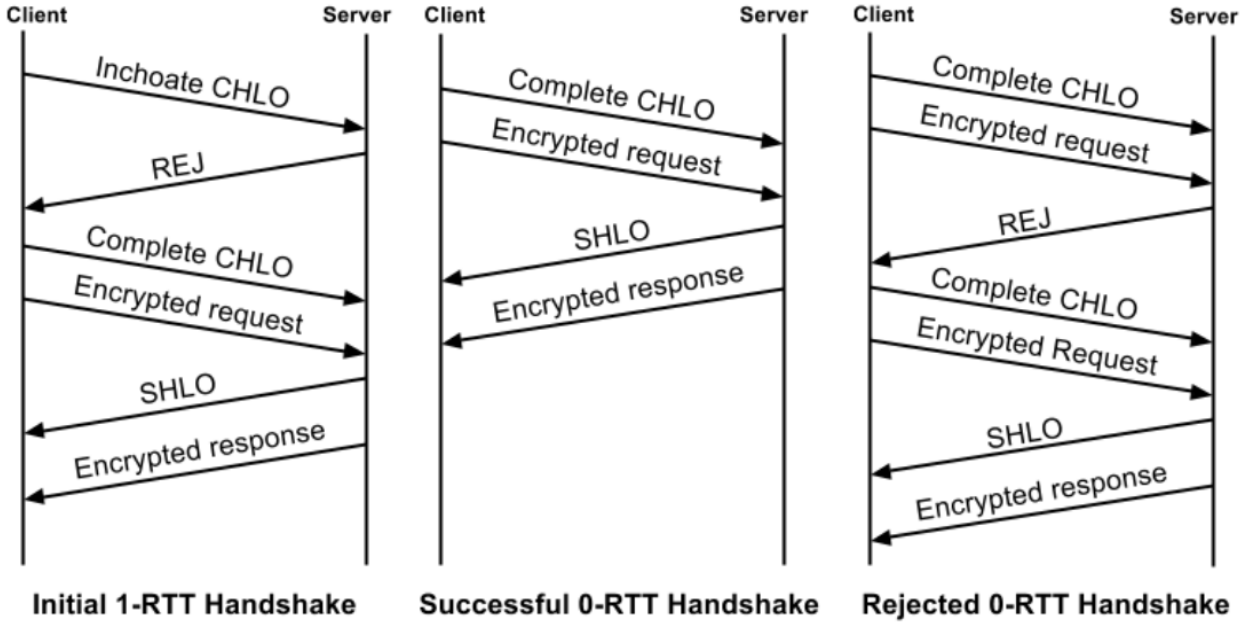


Figure 3.5.: Connection establishment in QUIC. Timelines of an initial handshake, an successful and un rejected 0-RTT handshake.
(Figure source: [36])

3.3.1 Initial Connection Establishment

When a client has no information about a server, which is true for the very first connection, it sends an inchoate client hello (CHLO) message to a server. The servers answer is a reject (REJ) message containing a signed server configuration, a certificate chain and a source address token [36]. Bases on the REJ content, the client then sends a complete CHLO containing its ephemeral cryptographic information.

This information may also be used for later fast connection establishment with a 0-RTT latency. The server answers with a server hello (SHLO) message containing own ephemeral cryptographic information which completes the handshake. Based on the ephemeral cryptographic information, both sides can calculate final keys, for encrypting subsequent data [36]. Figure 3.5 (left) depicts the timeline of the initial connection establishment in QUIC.
Note that the described handshake sequence is the fastest possible way of establishing an initial connection between a fresh pair of client and server. Based on the server's willingness to send a large proof of authenticity to an unvalidated IP address, the first REJ message may miss certain information. This may introduce several rounds of inchoate CHLO messaged and corresponding REJ messages.

3.3.2 Fast Connection Establishment

A client willing to perform fast connection establishment (i.e., 0-RTT data delay handshake) needs to have a server configuration that has been verified to be authentic. Fast connection establishment starts with sending a complete CHLO message, followed by encrypted data. A successful 0-RTT handshake is answered by the server with a SHLO message, followed by encrypted data. The timeline of a successful fast connection establishment is shown in Figure 3.5 (center).
In case the handshake fails (Figure 3.5 right), a REJ message is sent. This REJ message and the remaining handshake follows the scheme described in Section 3.3.1. The rejected initial CHLO message of the attempted 0-RTT handshake can be considered as inchoate CHLO message, resulting in a sequence similar to the initial handshake. Thus no additional delay is induced by a failing fast connection establishment. There may be several reasons for the rejection of a fast connection establishment. For example the source address token or even the server configuration may have expired. In some cases, the server may have changed its certificates, resulting in a failing fast connection establishment.

3.4 Controlling Packet Injection

After higher layer protocols have written data on a QUIC stream, the QUIC entity may divide this data into frames which in turn are bundled into packets. However before QUIC starts processing the higher layer data, this data has to pass the congestion control, as well as a two-level flow control.
Congestion Control: The used version of QUIC does specify a specific congestion controller. However in [36, 25] a reimplementation of TCP Cubic [24] is used. Moreover alternative congestion control approaches are currently under active research [25]. These approaches may involve `Congestion Feedback Frames`. The `Congestion Feedback Frames` are experimental and not used. However the intension is to provide extra feedback about the congestion state of a network path.
Flow Control: The QUIC flow control, which is very similar to HTTP/2's flow control, is composed of a two-level credit-based control mechanism. On connection level, the transmitted data of the entire connection is observed. Therefore the connection-level flow controller aggregates the bytes delivered and the highest received offset across all streams. The stream level flow control only observes the data sent on the individual streams, preventing a slowly draining stream from consuming the entire receiving buffer [36].
Both, connection- and stream-level flow control work in the same way. The flow control approach follows the sliding window and credit approach described in Section 2.1. A QUIC receiver is responsible for advertising the amount of data he is willing to receive. This advertising occurs in form of a window update, transmitted via a corresponding `Window_Update Frame`.

3.5 Security

QUIC is an fully encrypted and authenticated protocol, providing a security equivalent to TLS. With exclusion of early handshake packets and `Reset Packets`, the header of every QUIC packet is authenticated

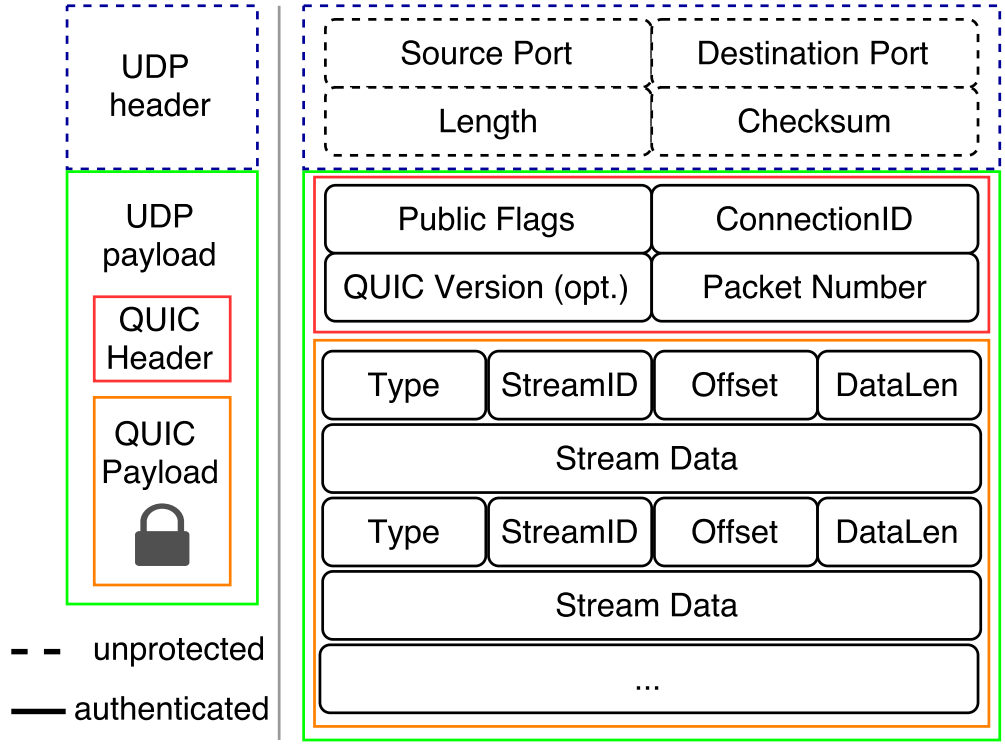


Figure 3.6.: Illustration of a QUIC packet inside an UDP datagram, consisting of a QUIC header and frames of (potentially) multiple, different streams. While the header is authenticated, the payload of a QUIC packet is also encrypted.

and the payload is also encrypted (Figure 3.6). Note QUIC's encryption and authentication mechanisms are specified in a separate document called QUIC Crypto [35]. As stated there, this document will be removed in the future and will be replaced by TLS 1.3 [12].

QUIC features a two-level security scheme. The first level is used for sending early client data, encrypted with initial keys. This keys are calculated with Diffie-Hellman values contained in the server configuration (as explained in Section 3.3.1). The security provided by the first level is similar to *Transport Layer Security (TLS) Session Resumption without Server-Side State* [36, 52] . For the encryption of server data and subsequent client data (second level), a `forward-secure key` (E_k) and a `Initialization Vector (IV)` are used. Based on ephemeral public values contained in the SHLO message, both sides calculate the forward-secure E_k and the IV. The keys are calculated using HMAC-based Extract-and-Expand Key Derivation [33, 7]. Both QUIC endpoints use the same key E_k and hold their own and the communication partner's IV. The forward-secure keys provide greater protection than the initial keys and are used for the current and subsequent connections.

Encryption and Authentication of Data: QUIC's full encryption of the payload and authentication of its headers corresponds to *Authenticated Encryption with Associated Data* (AEAD) [49]. The *Authenticated Encryption* (AE) refers to the payload and the additional *Associated Data* (AD) are the authenticated QUIC headers. Basically `Regular Packets` in QUIC are composed of a header followed by AEAD [25].

Encryption and authentication in QUIC is based on the *Advanced Encryption Standard - Galois Counter Mode* (AES-GCM) [53, 41, 11]. The general goals of AES-GCM are to encrypt the plaintext (PT), i.e., the QUIC payload, and authenticate the associated data, i.e., the QUIC header. For encryption, the PT is split into parts and each part is encrypted using AES separately. In AES-GCM, the AES encrypts a counter, which is increased for each part, with the key E_k and XORes the result with PT. To ensure that the ciphertext (CT) differs even if the same content is encrypted multiple times, the counter is initialized using the `Packet Number` and the IV.

To authenticate both, CT and AD, a *Keyed-Hash Message Authentication Code* (HMAC) [34] is used. Based

on a cryptographic hash function and the E_k, an *Message Authentication Code* (MAC) , covering the CT, the AD, the length of the CT, the length of AD and the IV, is calculated and transmitted with every regular QUIC packet. To verify the authentication of a QUIC packet, receivers calculate the MAC and compare it with the received MAC. After the authentication is verified, the decrypted content be calculated using E_k, the IV of the sender and the `Packet Number`.
Additionally QUIC's security specification addresses two different problems separately: *Address spoofing* and *Replay attacks*.
Address spoofing: Address spoofing [26] is a class of attacks where an attacker uses an address A, which is not its own address, to get a server to perform specific actions. Usually the server would perform this action only for the owner of A.
QUIC's answer to address spoofing is the *source-address token* (STK). The STK is an authenticated-encryption block created by the server which contains the address (i.e., IP address) of a client and an timestamp. Note that a STK with a particular IP is only send to a client with that specific IP inside the REJ message. From the clients point of view the STK is a opaque byte string. A client can reuse a STK for a later proof of the ownership of its IP address. However, the STK will only be accepted if the client did not move (IP address did not change) and if the STK is not expired. A server may only be willing to send its configuration if an enquirer provides a valid STK. Though this constraint may be relaxed to reduce latency. Other approaches, such as only requiring an STK when the amount of unknown connections with different IP addresses exceeds a limit, are conceivable.
Replay attacks: The phrase *replay* in the context of security means the capture of a message or a piece of a massage and use it at a later time [57]. Attackers can even replay encrypted messages to get servers to perform specific actions
To ensure that a message is fresh, peers generate a nonce and share it with their opposite. The receiver of a nonce has to include it for key derivation. Therefore replayed messaged are encrypted with an outdated key. Thus, the freshness of a message can be verified. However QUIC does not provide replay protection against the very first messages of a client prior to the first reply of the server. It is the applications responsibility to avoid sending replay attack vulnerable data until the server's nonce is available.

3.6 Deployability

A big issue in today's Internet is the deployability of new protocols or extensions of existing protocols. The monopole of the Internet Protocol (IP) on its layer is known and widely accepted. However in [50] a new waist of the Internet hourglass is described. With the high number of *Network Address Translations* (NATs) and firewalls, the UDP/IP and TCP/IP model shapes the possibilities of new transport protocol innovations.
Fundamental new transport protocol have a lower chance for wide deployment, as seen by MPTCP, SCTP and DCCP for example [27, 50]. The designers of QUIC understood that middleboxes are the key control points of today's Internet [36]. Instead of creating a fundamental new transport protocol, QUIC content is carried inside the UDP payload. Located there, QUIC is more likely to be accepted by middleboxes in the public Internet [50]. The already mentioned authentication of the QUIC headers (Section 3.1) further increases deployability by preventing middleboxes from manipulating content.
Another aspect, influence the deployability, is the implementation location of a protocol. Many existing transport protocols, e.g., MPTCP and SCTP, are implemented in the operating system (OS) kernel and thus require OS support. This slows down initial protocol adoption and innovations after deployment, in particular, as mobile devices are known to have long OS update intervals. In contrast QUIC is settled in the user-space, benefiting easy deployment. QUIC, its updates and potential extension may be shipped out without upgrading the entire OS. A wide-scale deployment of QUIC is possible as already shown [36].

We now propose a design of a multipath QUIC extension (MPQUIC). Based on the aspect that there are several multipath extensions for TCP (e.g., MPTCP, MMPTCP, MCTCP) but also fundamental new multipath protocols (e.g., SCTP), one can see that design space for multipathing support calls for many decision. A general architecture of the existing QUIC is illustrated in Figure 4.1 (left). Accordingly a MPQUIC extension (Figure 4.1, right) that uses multiple paths over a single connection is developed. Similar to MPTCP, the term subflow is introduced in MPQUIC. A subflow is a flow of QUIC segments, that operates over an individual path.

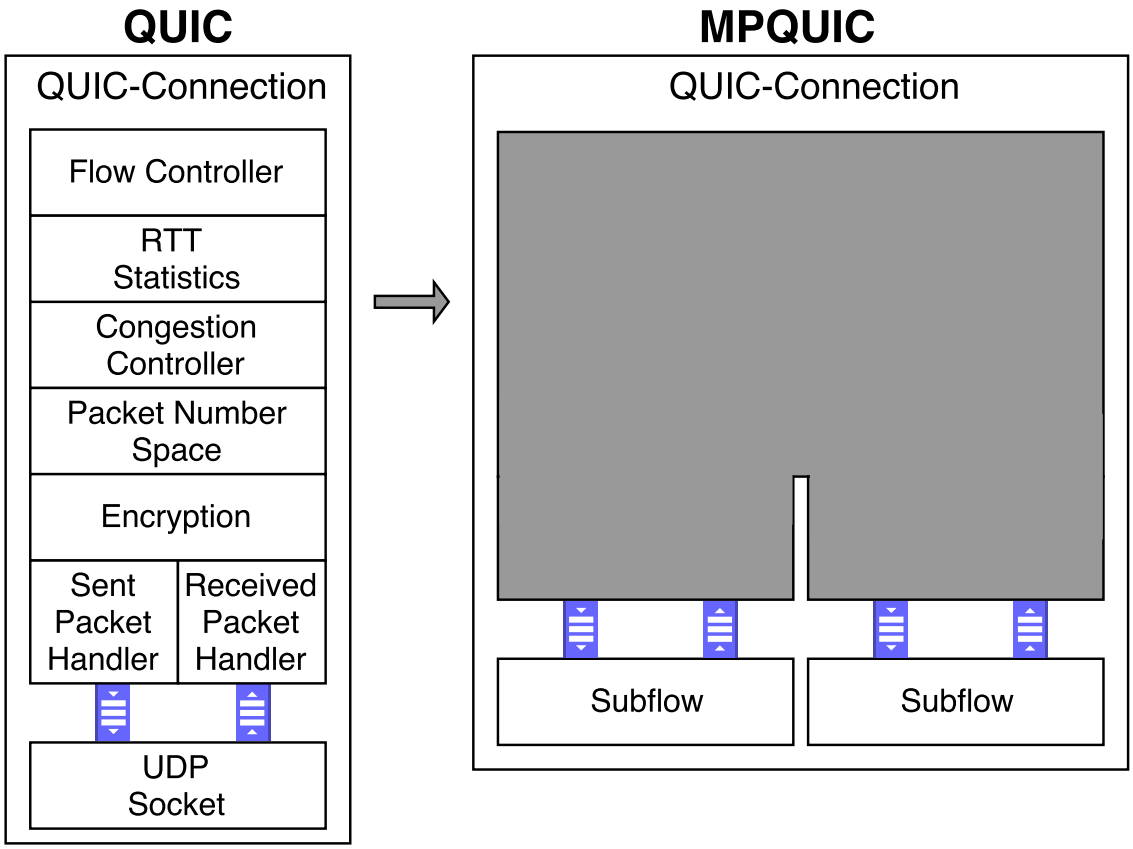


Figure 4.1.: The available design of QUIC uses a single UDP socket for transmitting data. The goal of MPQUIC is to extend QUIC to be multipath capable.

Starting with the requirements for a multipath QUIC extension up to scheduling decisions that arise later on, this chapter describes a general purpose MPQUIC design. Based on the single path QUIC architecture the MPQUIC architecture is established, and Figure 4.1 is extended accordingly for illustration. Note that this figure does not visualize individual streams, as they are already included in the design of traditional QUIC and not changed by MPQUIC.
At first, requirements for the design are defined. Next the Announcement of additional available paths is discussed. After specifying the packet number space, congestion and flow control, the acknowledging and retransmissions in MPQUIC is described. At last, encryption and the scheduling of packet packets is introduced.

4.1 Requirements for the Design

The design space for a multipath QUIC extension is very broad. There are many designs that may be qualified as a multipath extension, however there are also different advantages and disadvantages that will come up with individual design decisions. The objective here is to design a general purpose QUIC extension, that utilizes multiple paths while still being compatible with traditional QUIC. MPQUIC must able to communicate with traditional QUIC, without harming the single QUIC server or client. It is also desired that the additional protocol complexity of MPQUIC compared to QUIC increases as little as possible. The requirements according to this objective are defined as following:

- **R1** MPQUIC uses multiple paths
- **R2** MPQUIC is compatible with QUIC
- **R3** Additional protocol complexity of MPQUIC compared to QUIC is kept low

4.2 Establishing Additional Subflows

Before additional paths are used, the communicating QUIC entities must be aware of the available network interfaces. Assuming that one QUIC entity only knows its own network addresses, there needs to be a way to inform the communication partner about these addresses. For this purpose we introduce a process for establishing additional subflows, denoted as `Announcement`. We refer to the entity willing to announce its address as *Announcement Initiator* (AI). The receiver of the announced address is referred to as *Announcement Receiver* (AR).

Design of the Announcement

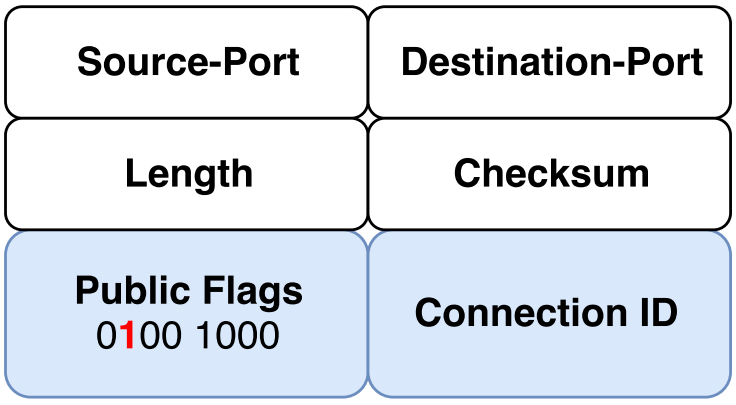


Figure 4.2.: UDP Header with `Announcement Packet` as payload: The QUIC header (blue) consists of the `Public Flags` and the `ConnectionID`. The `Public Flags` are in little-endian byte order. The `Multipath Flag` (0x40, red), used to identify the `Announcement`, is set to 1. The UDP header is visualized atop (white).

The AI has multiple network interfaces and wants to establish additional subflows for communication with the AR, thus the interfaces need to be announced. The required information for subflow establishment are the `ConnectionID` and the `IP/port` of the additional network interface. The corresponding `QUIC Connection` for subflow establishment is identified using the `ConnectionID`. The `IP/Port` tuple is used as destination address in the UDP/IP header when using the additional subflow. For sharing this information between AIs and ARs the`Announcement` is used.
For transmitting the `Announcement`, the initial established subflow or a subflow to be established may be used. A advantage when using a subflow to be established is that upon receiving the `Announcement`,

a receiver can be sure that at least one direction of the subflow is usable without middleboxes interfering. Additionally UDP/IP headers contain the IP/Port, required for the Announcement. Using this information in QUIC in combination with the ConnectionID in the QUIC header provides the AR with all required information for the Announcement. There is no need to carry the address in the payload or even create a new frame type for this purpose. Hence MPQUIC transmits Announcements on the subflow to be established.
Next ARs needs to recognize an Announcement as such. Therefore we propose a new packet type Announcement Packet which has a Multipath Flag set. Fortunately QUIC was originally intended (but is not specified) to support multipathing. A Multipath Flag in the Public Flag (i.e., bit 0x40) field of the QUIC header is already reserver, but not further specified, in the QUIC specification [25]. This benefits the design of MPQUIC, making it QUIC compatible and less complex. Announcement Packets look like regular packets without payload for traditional QUIC. While QUIC does nothing when receiving Announcement Packets, MPQUIC recognizes it as Announcement and extract the source address of the UDP/IP header. A complete Announcement Packet is visualized in Figure 4.2. For the meanings of the individual flags recap Table 3.1.
Note that the Announcement Packet type follows the concept of traditional QUIC's Public Reset Packet and Version Negotiation Packet. A single bit in the Public Flags field indicates how to interpret the unencrypted packet header while no payload is carried.

Announcement-Handshake

After designing the Announcement Packet, the transmission sequence for subflow establishment is investigated. Several approaches for the Announcement are possible. They differ in the amount of messages sent prior subflow usage (i.e., data transmission). It needs to be considered when it is safe to use subflows and what happens when subflow establishment fails. Subflow establishment may fail duo to various reasons (e.g., firewalls, NATs, broken physical path, AR is not multipath capable) resulting in loss of the transmitted data. Note that MPTCP, for example, uses the MP_CAPABLE option (Section 2.2.3) to detect the multipath capability of the AR during connection establishment. A similar concept may be added to MPQUIC in future work.

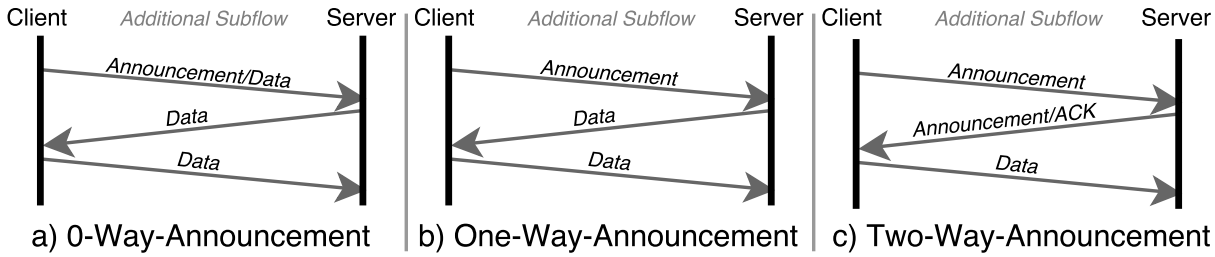


Figure 4.3.: Timelines for a) 0-Way-Announcement b) One-Way-Announcement and c) Two-Way-Announcement.

0-Way-Announcement: In QUIC 0-Way-Announcements are possible in theory duo to the connection identification based on ConnectionIDs. An Announcement Packet followed by data (Figure 4.3,a) is sent by the AI and matched to the corresponding connection by the AR. Upon sending and receiving the Announcement Packet respectively, the subflow is established. The data delay for the AI is 0-RTTs. The AR has to wait until the Announcement Packet is received and thus has a 1/2-RTT data delay.
The 0-RTT delay is very beneficial for multipath protocols duo to early subflow utilization, however it can only be used in QUIC when it is ensured that both the AI and the AR support multipathing. In scenarios where ARs are not multipath capable, the receiving of data on the new subflow is also successful, however connection migration is triggered. ARs executing connection migration discard information about the initial subflow (e.g., RTT estimations, congestion states) and only use the new subflow for further

data transmission. Thus the 0-Way-Announcement is not compatible with traditional QUIC, harming requirement R2.
Another drawback of 0-Way-Announcements is the higher chance of head-of-line blocking when subflow establishment fails. Data sent on an already established subflow, that depends on data sent in the 0-Way-Announcement, is blocked until the data has been retransmitted. Since there are no RTT-statistics for the new subflow, a suitable time for timeout and retransmission alarm can not be set. Therefore it is uncertain how long to wait until a packet is considered lost. Possible solutions are to send data redundant on both subflows or only send independent data in the 0-Way-Announcement. Both solutions add additional complexity to the protocol and reduce the advantage of a 0-Way-Announcement.
One-Way-Announcement: The One-Way-Announcement counteracts the drawbacks of the 0-Way-Announcement. In the One-Way-Announcement, AIs are not allowed to send data immediately after sending the `Announcement Packet` (Figure 4.3,b). Receiving ARs extract the address out of the `Announcement Packet`. They can be sure that the announced subflow is usable in one direction at least. Additionally ARs without multipathing support do not trigger connection migration since no data has been received. This counteracts some drawbacks of the 0-Way-Announcement, making it more resilience to subflow establishment failing and multipath incapable ARs. After receiving `Announcement Packet`, ARs may use the subflow for sending data, thus ARs have a 1/2-RTT delay. AIs may use the subflow after receiving the first bytes of data over the additional subflow. This corresponds to an implicit announcement confirmation and leads to a 1-RTT delay for the AI.
Nonetheless the problem regarding the missing RTT statistics remains. Also for early data, sent by ARs after 1/2-RTT, that needs to be retransmitted, duo to a subflow that is usable in one direction only, has no suitable timeout. Note that this risk is smaller compared to the 0-Way-Announcement since ARs can be sure that the subflow can at least be used unidirectional. Moreover a possible solution for the AR's RTT estimation is given in Section 4.7.
Two-Way Announcement: In the Two-Way-Announcement, ARs are not allowed to send data immediately after receiving the `Announcement Packet`. The Two-Way Announcement extends the One-Way-Announcement by one additional `Announcement Packet` sent by the AR, explicitly confirming the `Announcement` of the new subflow (Figure 4.3,c). When AIs receive the confirmation `Announcement Packet`, they can be sure that the subflow is usable in both directions. Thus AIs may send data over the new the subflow after 1-RTT delay, this equals the delay of the One-Way-Announcement.. Only after receiving data from the AI, ARs are sure that a subflow is usable in both direction. Thus ARs, have to wait one additional RTT (i.e., 1.5 RTTs), compared to the One-Way-Announcement until the subflow is used.
The obvious advantage of this approach is that both, the AR and the AI, only send data if the subflow is bidirectional usable. Moreover first RTT measurements are present when data is sent and more suitable loss detection alarms are set. On the contrariety, additional delay for data transmission is present. The possibility of fast subflow establishment, benefited through QUIC's design is not used.

We conclude that different designs for `Announcements` are possible. On one side, if it can be ensured that every entity supports multipathing, 0-Way-Announcements are conceivable. However there are various other reasons for subflow establishment failing. On the other side, if we see MPQUIC as an extension that needs to be compatible with QUIC (requirement R2) the drawbacks of 0-Way-Announcements are dominant. Two-Way-Announcements guarantee that data is only sent, when prior QUIC packets (i.e., the `Announcement`) have already been transmitted successfully. This guarantee costs one additional RTT data delay. The One-Way-Announcement is a trade-off between the 0-Way- and Two-Way-Announcement. While data is only sent, when both peers support multipathing, unidirectional subflows can still harm data transmission. However, ARs have the lowest possible data delay of 1/2-RTT on new subflows.
Especially beneficial are slow AR data delays for a typical client/server model, such as in HTTP. Devices owning multiple addresses (i.e., AIs) are usually clients (e.g., web-browsers of laptops, smartphones with WiFi and LTE), while servers (excluding datacenters) tend to have only a single Internet connection (i.e.,

ARs). Using the One-Way-Announcements, servers, in their role as AR, have a lower delay until new subflows can be used compared to clients. Further, in HTTP it is common that small HTTP requests are sent by clients. The servers typically answer with much larger HTTP response, containing the requested data. The One-Way-Announcement benefits this model by providing the lowest possible data delay to servers. Because all the defined requirements are achieved with this approach and the trade-off between early subflow usage and resilience to establishment failing is suitable for general purpose, we propose the One-Way-Announcement for the general MPQUIC design.

Earliest time of subflow establishment

The time when subflows are being established at the earliest, is an important aspect and determines the delay until subflows are usable for data transmission. We introduce and discuss subflow establishment a) beginning with the connection establishment b) during connection establishment and c) after connection establishment.

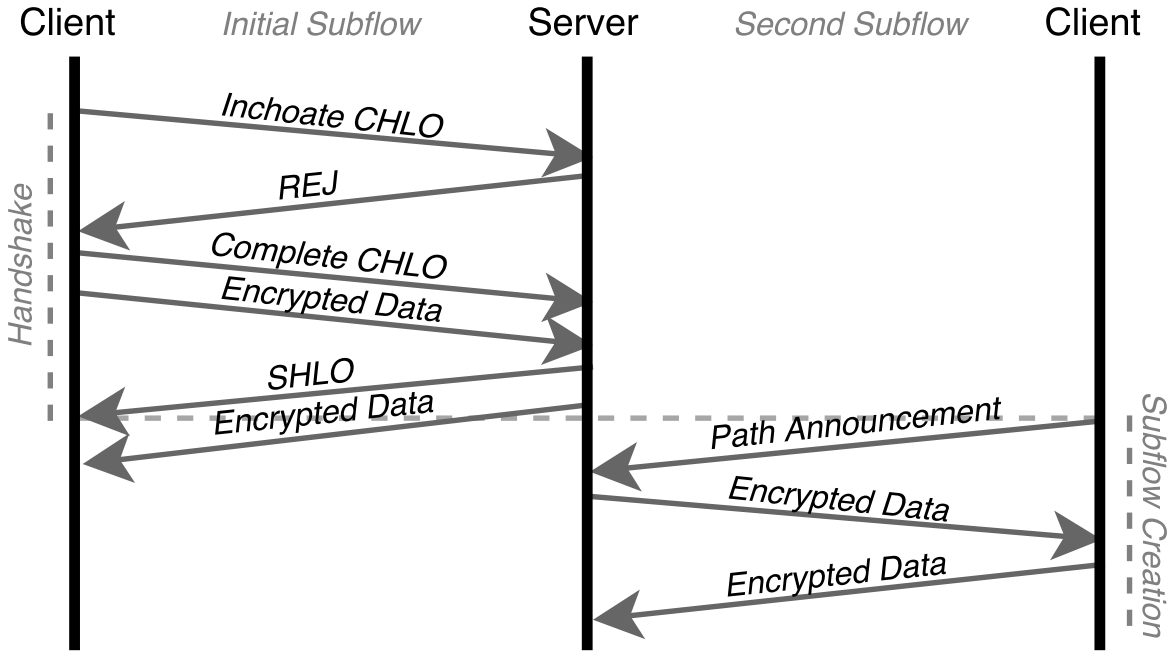


Figure 4.4.: Timeline of subflow establishment in MPQUIC. The left side depicts the initial QUIC connection establishment. The announcement of any additional subflow, which can be started after receiving or sending the SHLO respectively, is shown on the right side.

Beginning with the handshake: Here clients have no information about servers. Neither the server configuration nor the STK has been shared. Beginning the subflow establishment at the same time as the connection establishment is similar to an additional connection establishment on the additional path. Set `Multipath Flags` in the very first messages indicate that both endpoints are multipath capable. If a `Multipath Flags` is not set, a single path connection is used. Else the earliest established connection serves as initial subflow. To use both subflows for a single connection afterwards, there needs to be a *merge* mechanisms where both established connections need to agree on a single `ConnectionID` and security information and also combine their stream. One simple and fast merge approach is to discard one of the established connections. After the first connection is established, data is simply sent over both paths.
An advantage of this approach is that after connection establishment, the RTT of both subflows are

known and also the lowest RTT subflow is the initial connection. However, the protocol overhead and the implementation complexity is very high duo to two connection establishments while in the end only one connection is used. Therefore this approach is not further specified.

During the handshake: Subflow establishment during the handshake is possible in QUIC. After QUIC clients send the complete CHLO respectively QUIC servers receive the complete CHLO, QUIC connections may be used for data transmission and the `ConnectionID` is used to match the connection. Similar an `Announcement Packet` may be sent and matched using the `ConnectionID`. However there may be no connection to match because an additional subflow with a lower RTT than the initial subflow may cause an `Announcement Packet` to overtake the CHLO. This leads to a failing `Announcement` duo to a missing connection. In contrast to other approached, here it is important to retransmit the `Announcement Packet`. Again this time there is no RTT measurement for new subflows and thus no reasonable timeout for the `Announcements-Packet`, which makes it difficult to recognize failing `Announcements`.

After the handshake: Another alternative is to announce addresses after sending or receiving the SHLO respectively, that is after connection establishment. Compared to b), in c) there is an additional subflow establishment delay of 1-RTT for the client. In case the server is the AI, there is no additional delay. An advantage is that failed `Announcements` are attributable to the used path (e.g., middleboxes interference or packet loss) or to a multipath incapable AR, whereas in b) `Announcement Packets` may have overtaken the CHLO.

Summarizing the different points in time for subflow establishment, approach a) has a huge overhead and introduces unnecessary protocol complexity compared to b) and c). In b) there is an additional reason for subflow establishment failure. `Announcement Packets` can overtake CHLOs. The design needs to specify how to handle this case. Therefore we propose to establish the subflow after connection establishment (c). The protocol overhead and implementation is minimal compared to a) and b). If subflow establishment fails in c) it is reasonable to assume that the subflow is not usable. A complete `Announcement` is illustrated in Figure 4.4.

4.3 Combining, Separating, Mapping: The Packet Number Space

In QUIC, packet numbers are monotonically increasing unsigned numbers starting with 1. QUIC specifies a packet number to be unique for the entire connection. Every fresh or retransmit packet gains a packet number, higher than any packet number before. However this is designed for a connection with only a single subflow. For multiple subflows this concept may be adopted, i.e., a single packet number space for the entire connection. Each packet number occurs only once regardless of the subflow. The advantage of a combined packet number space is the comply with the traditional QUIC specification. Moreover ACK frames can be used for acknowledging packets of the entire connection. On the other side, ACK frames are also used for RTT estimation. Using ACK frames for the entire connection would require a proper design to not harm the RTT estimation. Another problem that occurs by using a combined packet number space is distinction between lost packets and late packets. When putting packets on two subflows with substantially different RTTs, receivers do not know if a missing packet is lost (scheduled on the lower RTT path) or just late (scheduled on the higher RTT path). This ambiguity requires longer timeouts for receiving packets before sending ACKs or sending data redundant duo to ACKs that signal a false positive missing packet.

A separate packet number space for each subflow solves the distinction problem. Each subflow starts with the packet number 1 and increases this number independently of the other subflows. The previous mention increasing unique packet number specification of traditional QUIC stays true for the individual paths.

In contrast, the established MPTCP extends the per subflow sequence numbers (SSN) by data sequence numbers (DSN). The DSN serves as a mapping among subflows used for in-order delivery. In QUIC the ordered delivery of application data is not based on packet numbers. Thus QUIC's natural design makes a mapping between the subflow packet numbers unnecessary.

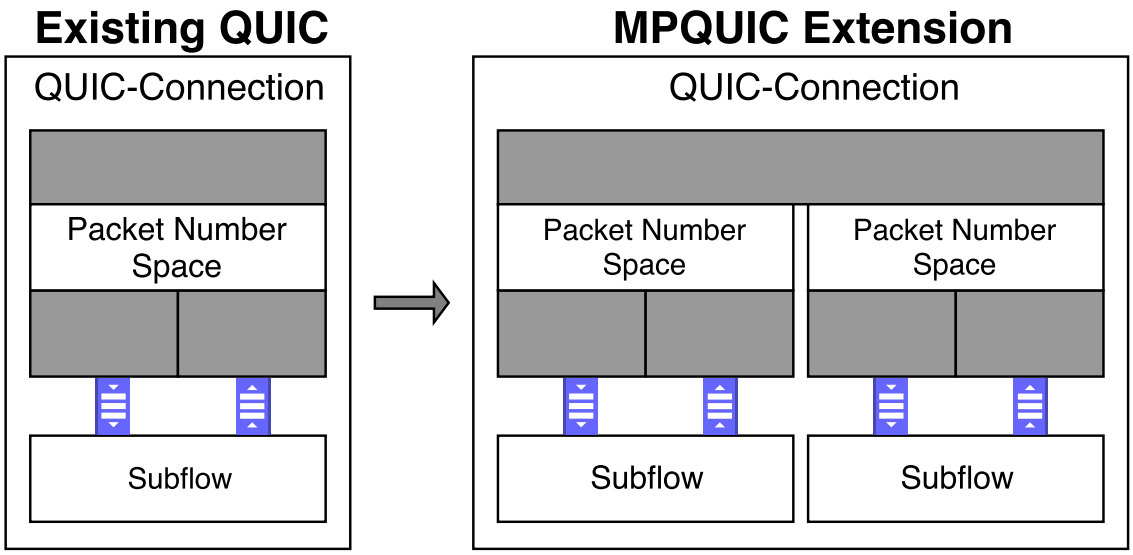


Figure 4.5.: In MPQUIC there is a separate packet number space for each subflow.

The separate packet number space in MPQUIC for each subflow is shown in Figure 4.5. Each of the individual packet number spaces exactly complies with the packet number space specification of traditional QUIC. Starting with a packet number 1, each subflow increases the packet number individually of each other, allowing the occurrence of the same packet number on different subflows.

4.4 Congestion Control Considerations

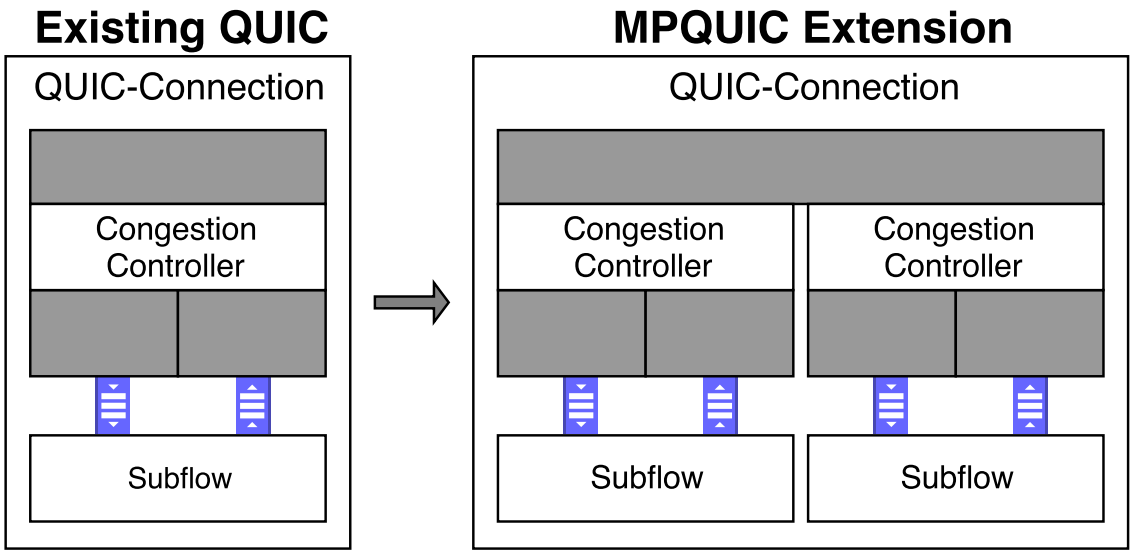


Figure 4.6.: A decoupled congestion controller in MPQUIC only observes and controls the data on its subflow.

Using additional subflows allocates additional bandwidth, assuming independent bottlenecks, that can be used by transport protocols. Congestion controllers must be aware of the available capacity and control how much data is sent to avoid congestion of network paths. As discussed in Section 2.1, the design of congestion controllers is very complex and under active research. Thus an optimal design for MPQUIC's congestion control is postponed for future work.
For a first general MPQUIC congestion control we propose a decoupled congestion control. Since we assume that the bottlenecks of the different subflows are at different places, it is reasonable to have a separate congestion window per subflow. Based on the received ACKs of individual subflows, the per subflow states of the congestion controller change and the per subflow congestion windows are adjusted.

The decoupled congestion controllers in MPQUIC are visualized in Figure 4.6. Each congestion controller maintains individual congestion parameters and phases per subflow. Similar to QUIC, a TCP Cubic is used for each subflow in use. Each Cubic has information only about the own maintained subflow and controls the amount of data to send on this subflow regardless fairness considerations.

4.5 Controlling the Flow

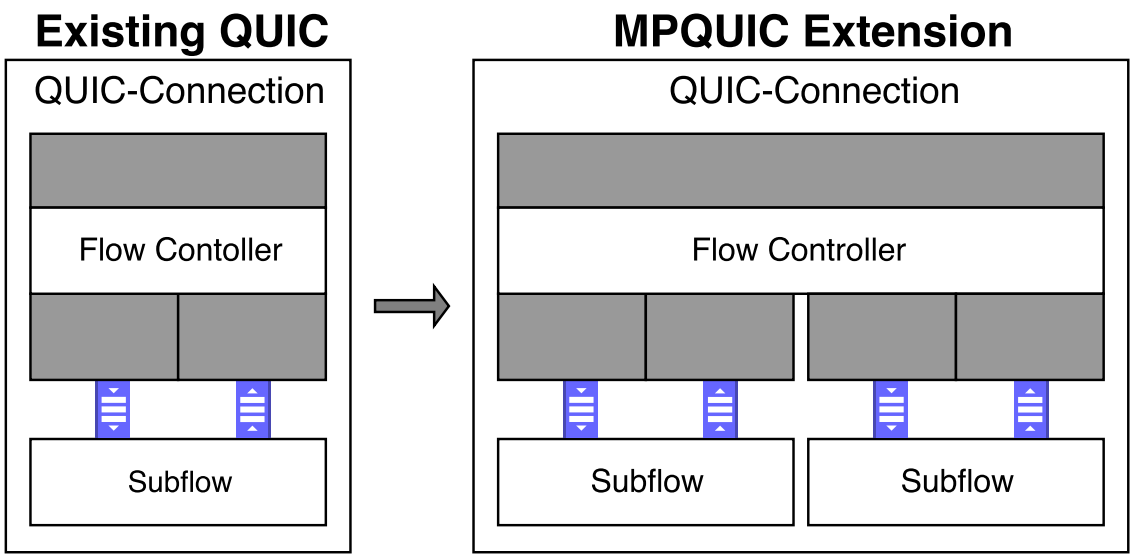


Figure 4.7.: Flow control in QUIC, as well as in MPQUIC, is on connection and on stream level. There are no architectural differences in both versions.

A quite different behavior and design can be observed when examining the flow controller. The bottlenecks considered by flow controllers are the receivers buffers. It is possible to extend the current flow control by one additional level, the subflows. A resulting three-level flow controller observes the connection, each stream of a connection and also each subflow of a connection. Another approach is to keep QUIC's two-level flow control. The connection and streams are still observed while the existence of individual subflows is ignored.
The selection of the flow controller design depends pretty much on the receiver buffers of a particular QUIC or MPQUIC implementation. A receiver may either have buffers for each subflow (i.e., multiple buffers per connection) or a single buffer per connection. For a general MPQUIC, we assume that there is a single buffer per connection and thus the flow controllers ignore subflows.

The subflow independent flow controller of MPQUIC is illustrated in Figure 4.7. Following the concept of QUIC, the stream- and connection-level flow control remains untouched and is adopted to MPQUIC. There is no subflow specific flow control. As a result only the flow of the individual streams and the connection is controlled.

4.6 Reliability: Acknowledgements and Retransmissions

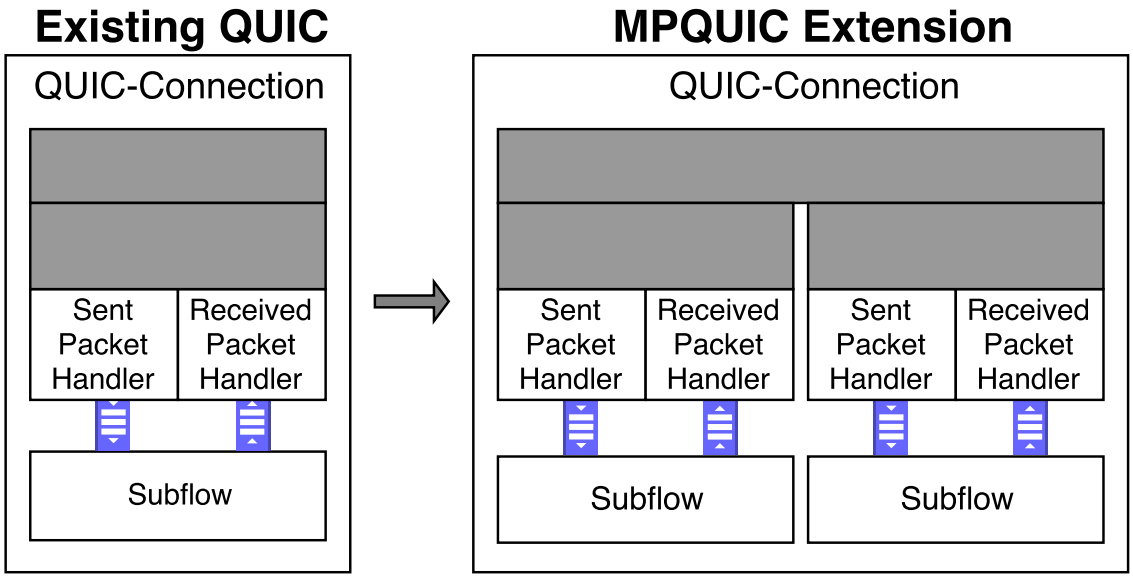


Figure 4.8.: The sent and received packet handlers, responsible for lost detection and acknowledging, exists once per subflow.

Traditional QUIC is a reliable transport protocol. Thus received and missing packets are recognized by a receiver and acknowledgements are sent. MPQUIC uses multiple subflows over potential different network paths that have their own network characteristics. To provide reliability equal to traditional QUIC, the acknowledgements and retransmissions need to be adapted for multipathing.
In general there are two different possible approaches for acknowledging. The first is to acknowledge received packets for the entire connection. Acknowledgements may be sent over a subset of the established subflows. For example, the subflow with the lowest RTT may be used for all acknowledgements.. Fine-grained optimizations or application specific scheduling are further benefited by this approach. On the other side, as already mention in Section 4.3, the calculation of the RTT is based on acknowledgements. Furthermore duo to the MPQUIC packet number space, which is subflow specific, a mapping between the acknowledgement and the packet number space is required. Therefore a combined acknowledgement space requires several protocol adjustments and is not well suited for the current design of MPQUIC.
The most reasonable approach for acknowledging are subflow dependent acknowledgements, according to the congestion control, packet number space and RTT estimation. Similar to traditional QUIC, there are acknowledgements and a loss-detection mechanism, including alarm timers, for each subflow. The advantage of this design is its conformance with the packet numbers and congestion control and its simplicity duo to very low protocol complexity compared to QUIC.
In QUIC we distinguish mechanisms for acknowledgements and retransmissions. The receiving of packets is acknowledged using ACK frames, informing about missing and received packet numbers. For

retransmission, only the payload of lost packets is considered. The frame and packet boundaries of the lost data are irrelevant upon loss. Upon retransmission the data will be framed again and packet into new packets. This allows MPQUIC to schedule retransmission data onto an arbitrary subflow which enables further optimization. For a general MPQUIC design, the data for retransmission is not special treated and scheduled like fresh data.

Sent- and received packet handlers, depicted in Figure 4.8, are responsible for loss detection and acknowledging. To achieve the same reliability in MPQUIC as in QUIC, the loss recovery mechanism of a single subflow is duplicated for every additional subflow. Retransmission queues, alarm timers and acknowledgements exist per subflow and are independent to the remaining subflows. However the retransmission of lost data can occur on an arbitrary subflow. In general retransmission data is treated like fresh data.

4.7 Dealing with multiple Round-Trip Time statistics

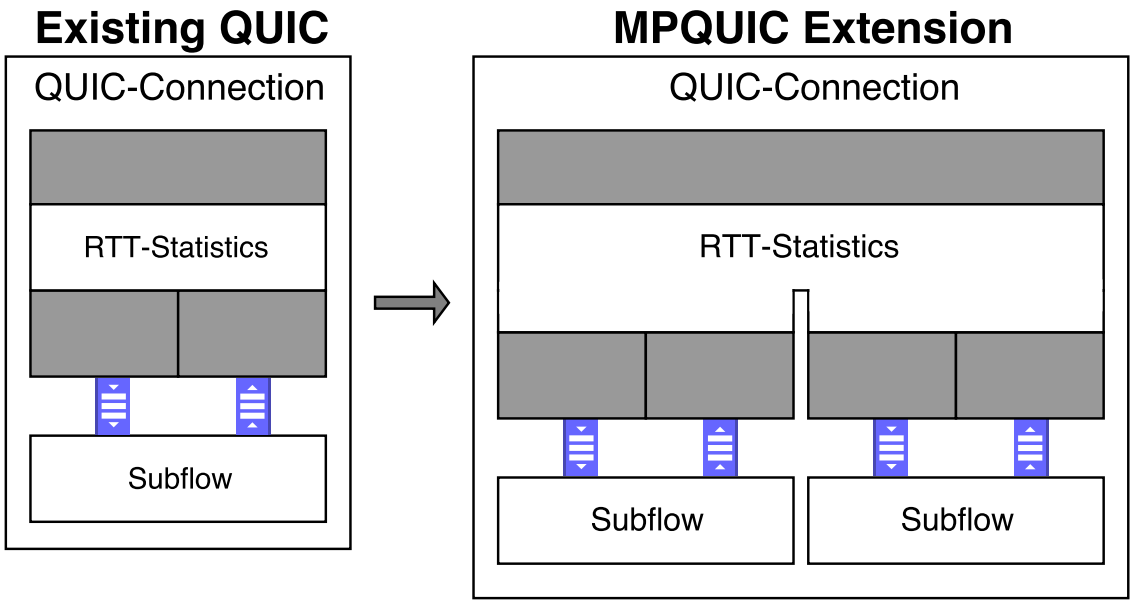


Figure 4.9.: In MPQUIC, the congestion control and loss detection requires subflow specific RTT-statistics. The flow control requires derived subflow overreaching information.

QUIC uses statistics about the RTT (RTT-statistics) for various components (i.e., flow control, congestion control, lost-detection). The RTT-statistics (e.g., minimal RTT, maximal RTT, smoothed RTT) are derived information about the RTT of a connection. Most of this information refers to a single subflow. For a multipath extension it has to be distinguished between components that require per path RTT-statistics or derived path-broad RTT-statistics. Based on the design of the individual components, the RTT-statistics are adjusted accordingly. The affected components are i) the loss detection, i) the congestion control and iii) the flow control.
Detecting lost packets (i) is a path depended issue, as described in Section 4.6, and based on timeouts. These timeouts are determined by the RTT. Since different paths have different latencies, per path RTT-statistics are used for subflow specific lost detection alarm timers.
A congestion controller (ii) exists once per subflow (Section 4.4) and therefore requires per path information about the RTT to determine the path depended congestion state.

The flow controller (iii) in MPQUIC operates on connection and stream level. There is no subflow-level flow control (Section 4.5). Hence, RTT-statistics used by the flow controller are derived and combined from the per path RTT-statistics. Based on the smoothed RTT a QUIC flow controller determines the offset of the window updates (the amount of data which can be sent) and its sending interval. Here, the MPQUIC flow controller makes use of the minimum smoothed RTT of the individual paths resulting in smaller offsets and shorter intervals. The intension of the minimum is that window updates are sent and adjusted with respect to the lowest (smoothed) RTT subflow. When assuming the low RTTs enable higher throughput, taking the minimum smoothed RTT is reasonable for throughput maximization, even in networks with substantially different subflow RTTs.
Note that other approaches need to be evaluated in future work to obtain an optimized flow control. We suggest three derivations of the subflow specific smoothed RTT statistics for a MPQUIC flow controller:

1. The minimum smoothed RTT over all paths. Taking the minimum may result in smaller window increments and more window updates.
2. The maximum smoothed RTT over all paths. Taking the maximum may result in larger window increments and less window updates.
3. A combination of all smoothed RTTs (e.g., mean, median). Taking a combination may result in a moderate number of window increments and window updates.

RTT on a new Subflow

When creating new subflows using One-Way-Announcements (Section 4.2), no RTT estimations are available for Announcement Receivers (ARs). For RTT estimation usually the delay between sending data and receiving the acknowledgement is used. In the One-Way-Announcements the subflow establishment is not being acknowledged but immediately used for data transmission. However some schedulers as well as the loss detection rely on RTT measurements. Especially the loss detection that uses alarm timers, based on RTTs, has no appropriate way to decide if data on new subflows has a high delay or is lost.
For subflow creation in MPQUIC, a new method to obtain a very first RTT estimate is required. Fortunately when creating new subflows, a connection and thus established subflows are available. When sending a packet on an established path parallel to an `Announcement`, the difference of the receiving time of both packets may be used by the AR for a first RTT estimate of the new subflow.
For MPQUIC we propose a *between-subflow delay based RTT estimation* for One-Way-Announcement based subflow establishments. When announcing an additional address using an `Announcement` on the new subflow, the AI also marks the next packet, denoted in the following as *EP (Estimation Packet)*, on any established subflow with the `Multipath Flag`. Here it is assumed that both packets, `Announcement` and EP on the different subflows are sent at approximately the same time (send time $s_0 =$ send time s_1). An AR receiving the EP on an established subflow (S_0 with RTT_0), utilizes this packet's receiving time, r_0, for estimating the RTT of the new subflow (S_1 with RTT_1). The `Announcement` is received at r_1. Note that additional delays (e.g., queuing, differences in s_1 and s_0) are not considered in this approach as their impact is negligible low. Further in case the EP is received after the `Announcement` (i.e., $r_0 > r_1$), the AR would have to wait until receiving the EP. For a fast approximation we suggest to use $RTT_1 := RTT_0$ if $r_0 \geq r_1$, to obtain a RTT estimation immediately. According to $r_0 \geq r_1$ we expect that $RTT_0 \geq RTT_1$ and thus RTT_0 is a upper bound for RTT_1. For schedulers as well as alarm timers, setting RTT_1 equal RTT_0 is reasonable for a very first estimation.
For estimating RTT_1 if $r_0 < r_1$, we assume that the one-way delay D is the same in both directions and thus RTT equals twice the one-way delay D of a subflow.

$$RTT = D \times 2 \tag{4.1}$$

The delay D of a subflow is the difference of the receiving time (r) and sending time (s) of a packet.

$$D = r - s \tag{4.2}$$

Using Equation (4.1) and Equation (4.2) gives us a equation for calculating the sending time.

$$RTT = D \times 2 = (r - s) \times 2 \quad \rightarrow \quad s = r - RTT \times \frac{1}{2} \tag{4.3}$$

We further use the previously assumed equality of the `Announcement` sending time s_1 and EP's sending time s_0.

$$s_1 = s_0 = r_0 - RTT_0 \times \frac{1}{2} \tag{4.4}$$

Now RTT_1 can be estimated using Equation (4.1), Equation (4.2) and Equation (4.4).

$$RTT_1 = D_1 \times 2 = (r_1 - s_1) \times 2 = (r_1 - s_0) \times 2 = (r_1 - (r_0 - RTT_0 \times \frac{1}{2})) \times 2 \tag{4.5}$$

Therefore the RTT of the new announced subflow RTT_1 can be estimated using the receiving time of the `Announcement` r_1, the receiving time of the EP r_0 and the already existing RTT of EP's subflow RTT_0:

$$RTT_1 = (r_1 - (r_0 - RTT_0 \times \frac{1}{2})) \times 2 \tag{4.6}$$

For illustration an academical example is given next:
Let's have a WiFi subflow with $RTT_{WiFi} = 20ms$ and a LTE subflow with $RTT_{LTE} = 60ms$. While the WiFi subflow is the initial subflow and thus the RTT is already estimated, the LTE subflow is being announced and therefore RTT_{LTE} is unknown.
At $s_{LTE} = 500ms$ an `Announcement` of the LTE address is sent by the AI. At the same time $s_{LTE} = s_{WiFi} = 500ms$ an packet including some application data is send on the WiFi subflow tagged with the Multipath flag. Note that s_{LTE} and s_{WiFi} is unknown for the AR.
The AR received the EP at $r_{WIFI} = 510$. The announcement is received at $r_{LTE} = 530$. Using the measured r_{WIFI}, r_{LTE} and the already estimated RTT_{WiFi}, the AR now estimates $RTT_{LTE_{estimate}}$:

$$RTT_{LTE_{estimate}} = (530ms - (510ms - 20ms \times \frac{1}{2})) \times 2 = (530ms - 500ms) \times 2 = 60ms \tag{4.7}$$

This simple example showed the correctness of the approximation e.g., $RTT_{LTE_{estimate}} = RTT_{LTE}$, when neglecting other delays.

Remarks: An AR receiving the estimation packet EP processes the payload like any other packet. The only difference between an EP and a normal packet is the usage of the receiving time for a RTT estimation. An multipath incapable QUIC simply ignores the `Multipath Flag` and processes the packet as usual.

4.8 Encryption

The cryptographically information is shared in QUIC during the initial handshake. For forward-secure protection ephemeral keys are used. These keys may be used for fast connection establishment without generating new keys. Here establishing an additional subflow is treated similar to fast connection establishment. There is no additional key generation or exchange when additional subflows are announced. Figure 4.10 shows the subflow independent encryption in MPQUIC.
Based on the packet number, an initialization vector and the plaintext, QUIC packets are encrypted and authenticated. This usually prevents that encryption of identical plaintexts result in identical ciphertexts. However in MPQUIC each subflow has its own packet number space and thus a packet number is only unique with respect to each subflow. As the IV is used for the entire connection and packet numbers may appear multiple times, it is possible that encrypting identical plaintexts, sent over different subflows with the same packet number results in identical ciphertexts. Since this case is assumed to be rare we postpone a solution (e.g., random packet number initialization for new subflows, separate IVs for each subflow) for future work.

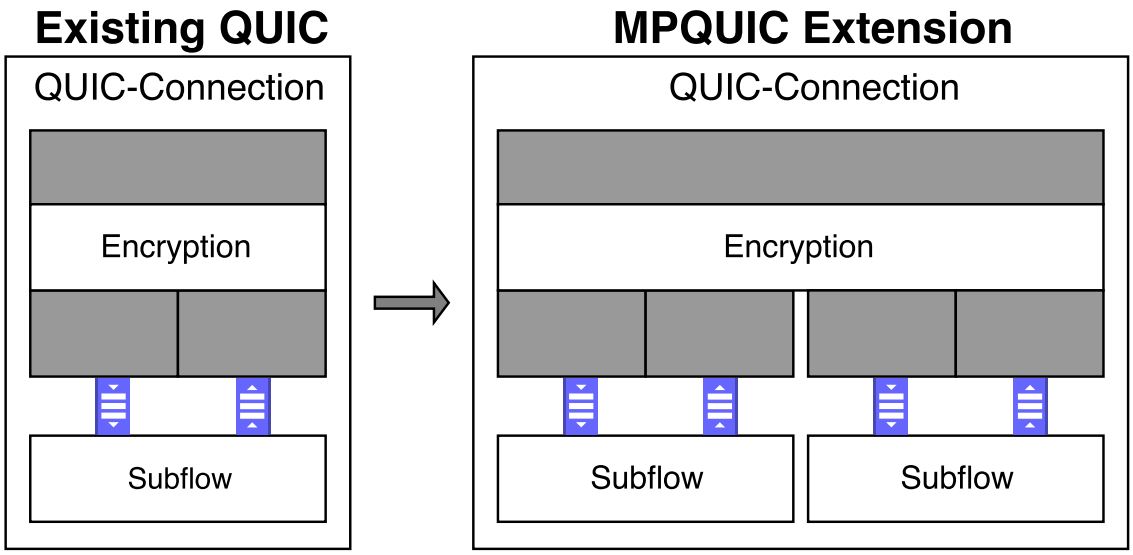


Figure 4.10.: For encryption of the data, MPQUIC uses the same cryptographically information for all subflows.

4.9 When to use which subflow: A Stream to Subflow Scheduler

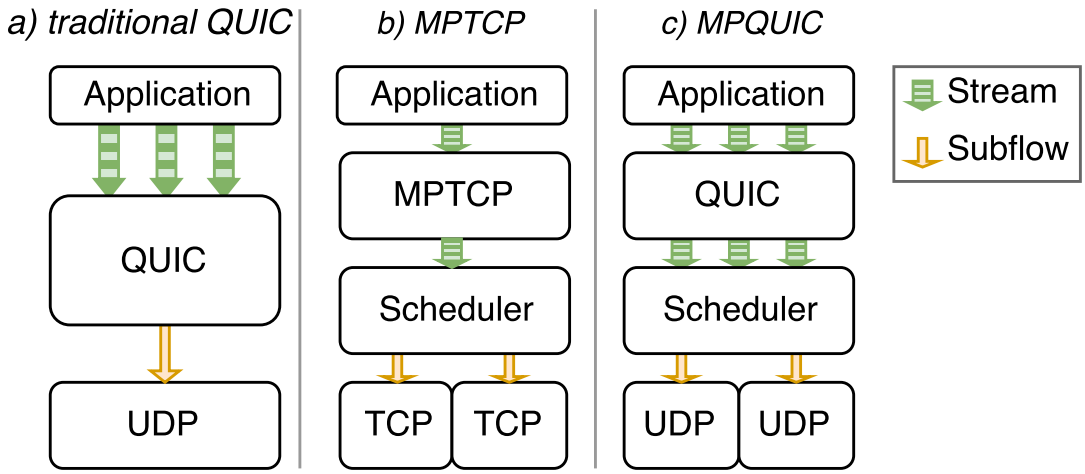


Figure 4.11.: Network stack comparison of QUIC, MPTCP and MPQUIC. QUIC multiplexes application streams on a single UDP flow (i.e., m:1), whereas MPTCP splits a single stream on multiple TCP subflows (i.e., 1:n). MPQUIC combines both features by multiplexing application streams on multiple UDP subflows (i.e., m:n).

The availability of an additional subflow requires to decide when to use which subflow. While in a single path setup, the decision becomes trivial, i.e., always use the one and only subflow, in a multipath setup, the scheduling decision depends on various aspects (e.g., preferences, subflow characteristics). As discussed in Section 2.4, there are many possible approaches for constructing and optimizing a scheduler. Especially the multiplexed streams, provided by QUIC, in combination with multiple subflow in MPQUIC calls for further optimization duo to the possibility of even more fine-grained scheduling. The scheduler in multipath QUIC, for example, has to ensure that a subflow has not exhausted its congestion and

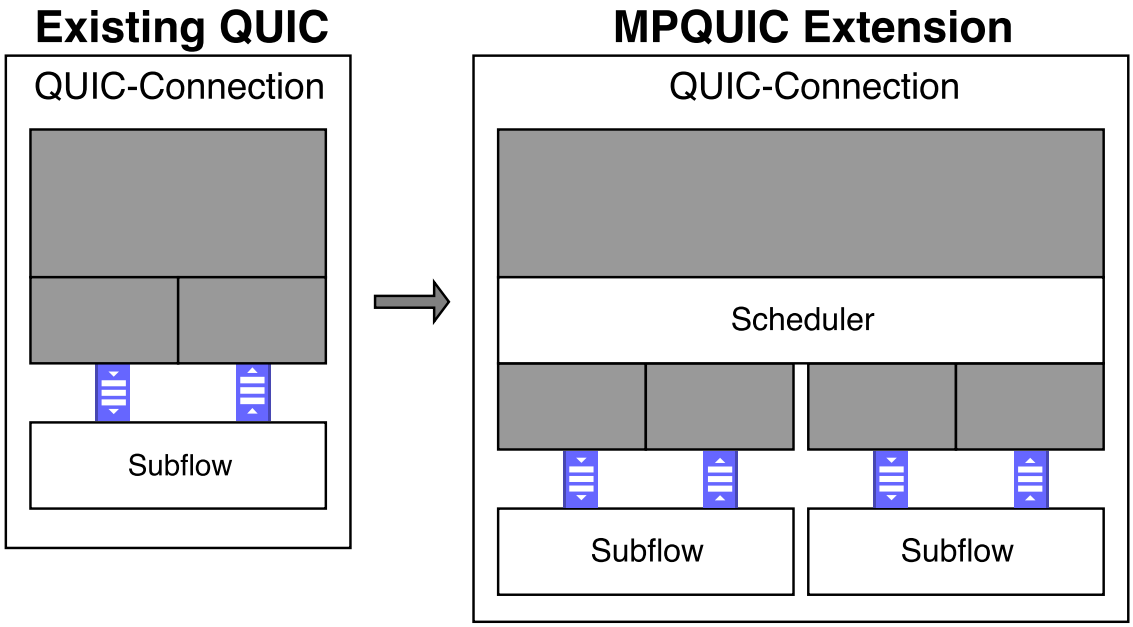


Figure 4.12.: The usage of multiple subflows requires a scheduler that determines when to send data on which subflow.

flow control window and the stream has not saturated its flow control window. Compared to MPTCP, where data of a single stream is scheduled onto *n* subflows (i.e., 1:n scheduling), MPQUIC allows to precisely select data of *m* individual streams and schedule this data on *n* subflows (i.e., m:n scheduling). Accordingly subflow properties as well as stream preferences are considered for the scheduling decision in MPQUIC. Moreover MPQUIC enables retransmissions scheduling on arbitrary subflows as mentioned in Section 4.6, e.g., stream data of lost packets might be combined to new packets together with fresh stream data on any subflow. A network stack comparison with highlighted stream and subflow awareness of QUIC, MPTCP and MPQUIC is given in Figure 4.11
However, the design and fine-grained optimization of a MPQUIC stream-to-subflow scheduler is out of this document's scope and MPQUIC's default subflow scheduler relies on a established solution. Based on previous work with MPTCP scheduling [19], we propose the Lowest-RTT-Subflow-First (Section 2.4.1) scheduler for MPQUIC, which is also the default scheduler of the Linux Kernel MPTCP implementation [43]. Stream scheduling is application dependent and also requires preferences or priorities of individual streams. Therefore default stream scheduling follows a simple RoundRobin scheme. Here the stream scheduler iterates over each stream, filled with data, and hands the data over to the subflow scheduler. We further envision a rich variety of scheduling flavors to consider stream and subflow preferences, e.g., to optimize retransmission handling in accordance with these preferences.

For multipathing, QUIC is extended by a single scheduling instance which schedules packets from streams on subflows (Figure 4.12) Thus, the MPQUIC scheduler composes packets out of the queued stream data and assigns these packets on subflows. This enables a fine-grained scheduling, considering both subflow properties and stream requirements. The default scheduler is the Lowest-RTT-Subflow-First Scheduler, which schedules packets on the subflow with the lowest RTT that is not congestion blocked.

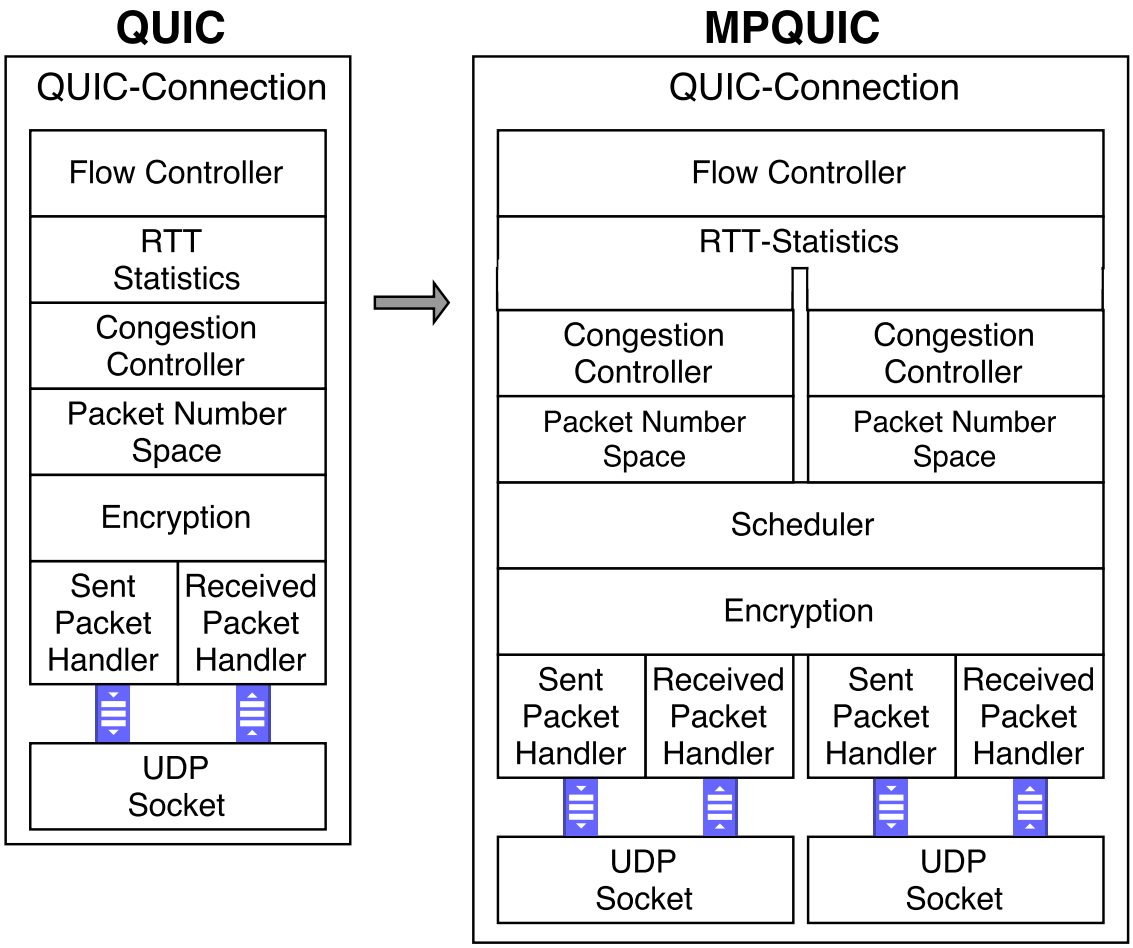


Figure 4.13.: MPQUIC's overall design compared to existing QUIC.

We presented our multipath extension of QUIC, illustrated in Figure 4.13, consisting of a One-Way-Announcement where an `Announcement Packet` is sent after connection establishment. After successful subflow establishment, a new subflow obtains its own packet number space and is observed by its own congestion controller. Lost packet detection is performed on subflow-level as well. Accordingly acknowledgements are created and interpreted subflow dependent. Data for retransmission is treated as fresh data. Subflow overreaching connection- and stream-level flow controllers observe sent data subflow independent. A Lowest-RTT scheduler finally schedules packets on the subflow with the lowest RTT while respecting the flow and congestion control.
The objective of the presented design is to serve as a general purpose multipath protocol, that is also able to communicate with traditional QUIC. Several design decisions, e.g., decoupled congestion control and no subflow-level flow control, are based on assumptions about the most common use case. Probably a receiver with buffers for each subflow is better served with an additional subflow-level flow control. Nonetheless this is not the general case. Duo to the large design space and optimization possibilities of certain components, i.e., the congestion controller and the scheduler, general solutions are used. The congestion control does not consider fairness, but tries to maximize per subflow throughput. In case of the scheduler, a already established solution, known from MPTCP, is chosen.
Overall traditional QUIC's design allowed easy development of our multipath extension. QUIC concepts, such as fast connection establishment, using the `ConnectionID` are very beneficial for MPQUIC subflow establishment, enabling subflow usage after only a half RTT. The new special packet type `Announcement`

Packet is similar to the other special QUIC packets and uses the already reserved Multipath Flag. Moreover we expect that the new stream to subflow (m:n) scheduling possibilities, described in Section 4.9, provide a huge advantage over the existing multipath solution MPTCP. Compared to MPTCP, which treats all incoming application data as a single stream in a FIFO order, MPQUIC distinguishes between individual data streams and thus enables the usage of application preferences. Further the single stream interface, provided by MPTCP adds dependencies between application data on the entire connection, whereas MPQUIC has dependencies only within streams. An example setup with a packet of small urgent data, that is delivered by MPQUIC without dependencies is illustrated in Figure 4.14.

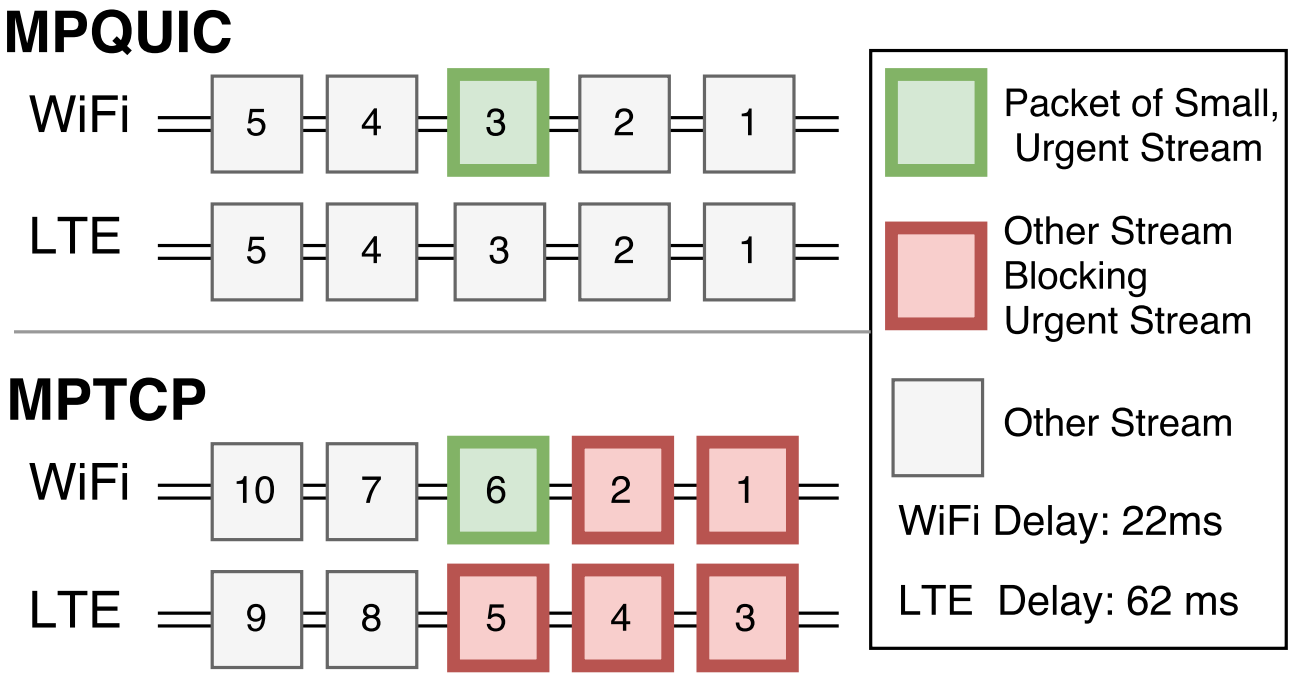


Figure 4.14.: Illustration of packets (including packet numbers) of different streams and subflows. In MPQUIC, former packets containing other streams, do not block the urgent stream. MPTCP's strict in-order delivery causes packets of other subflows to block the urgent stream.

Summary of the design

`Announcement`

The `Announcement` consists of an initial `Announcement Packet`, sent from an Announcement Initiator AI to an Announcement Receiver AR. The AR implicitly confirms the `Announcement` by sending data. Upon receiving this data, the AI may use the subflow for sending data as well. The earliest time an `Announcement` is possible is after connection establishment. The AI begins the `Announcement` after the SHLO is sent or received respectively, depending on the perspective (i.e., client or server) of the AI.

`Announcement Packet`

The `Announcement Packet` is a new packet type created for transmitting the `ConnectionID` and the `IP/Port` used for new subflows. The `ConnectionID` is included in the QUIC header. The UDP/IP header contains the `IP/Port`. To distinguish the `Announcement Packet` from other packets, the `Multipath Flag` is set.

`Separate Packet Number Space`

MPQUIC uses a separate packet number space for each subflow. Each of the individual packet number spaces exactly complies with the packet number space specification of traditional QUIC. Starting with

the packet number 1, each subflow increases the packet number independent of each other, allowing the occurrence of the same packet number on different subflows.

`Decoupled Congestion Control`

MPQUIC implements one congestion controller with individual congestion parameters and phases per subflow. That means a CUBIC congestion controller is used for each established subflow. Each CUBIC has information about the own maintained subflow only and controls the amount of data to send on this subflow.

`Two-Level Flow control`

QUIC's stream- and connection-level flow control remains untouched and is adopted to MPQUIC. There is no subflow specific flow control. As a result there are only stream and connection flow controllers that overreach subflows.

`Reliability`

To achieve the same reliability in MPQUIC as in QUIC, the loss recovery mechanism of a single subflow is duplicated for every additional subflow. Retransmission queues, alarm timers and acknowledgement frames exist per subflow and are independent to the remaining subflows. However the retransmission of lost data may occur on an arbitrary subflow. In general retransmission data is treated like fresh data.

`Lowest-RTT-Subflow-First Scheduler`

The MPQUIC design provides a single scheduling instance which schedules packets from streams on subflows. Thus, the scheduler composes packets out of the queued stream data and assigns these packets on subflows. This enables a fine-grained scheduling, considering both subflow properties and stream requirements. The default scheduler is the Lowest-RTT-Subflow-First Scheduler, which schedules packets on the subflow with the lowest RTT that is not congestion blocked.

5 Implementation of MPQUIC

The implementation of a MPQUIC prototype, based on the previous introduced MPQUIC design, is described in this chapter. The prototype is later used to proof the concept of MPQUIC and further evaluations. Therefore it is desired to have this prototype as soon as possible. With a fast available prototype, the strength of MPQUIC are demonstrated on one side, and on the other side it is possible to reveal potential weak spots in the design and revise concerned components. Therefore highly optimized performance is not as important as a shortly available MPQUIC prototype, showing the behavior MPQUIC. Moreover for future research, with a special focus on stream to subflow scheduling, it must be possible to easy modify existing schedulers and include fundamental new schedulers for MPQUIC. Therefore a *Plug and Play Scheduler* concept is introduces.

5.1 Available QUIC implementations

The usage of an existing QUIC implementation is a possibility to obtain a MPQUIC prototype shortly. In case a modifiable and well documented open-source QUIC implementation is found, MPQUIC may be developed as extension and thus avoid the implementation of the entire protocol. However while writing this document, there is neither a stable draft nor a stable implementation of QUIC. The existing implementations have individual focuses on certain aspects and therefore may be more or less suitable for our purpose.

Chromium QUIC[1]: The open source browser Chromium provides a QUIC implementation developed by Google employees. Additionally to the Chromium browser that utilizes QUIC, an example QUIC-Client and QUIC-Server is provided. The implementation is written in C++ and includes functionality tests. An advantage of Chromium QUIC is the development by Google. It follows and is compatible with Google's QUIC implementation of the Chrome browser. However the source code of QUIC is part of the chromium code base which makes it more difficult to modify. The Chromium dependencies is a essential drawback of this implementation.

proto-quic [2]: Proto-quic is a stand-alone C++ library and contains a subset of the Chromium source code. The intention is to provide the Chromium QUIC code without the need of all the Chromium dependencies. The supported platforms are a strict subset of the platforms supported by Chromium. The only supported platform is currently Linux.

Proto-quic is not an official supported Google product but is developed by some of the Chromium QUIC developers. At first sight the proto-quic implementation serves as a reasonable basis for MPQUIC. However proto-quic is not further updated and will be deleted in the future as stated on proto-quic GitHub.

goquic[3]: Goquic is a QUIC implementation by devsisters, a Korean company. It is implemented in Go and based on devsisters Libquic library, which again is based on Chromium QUIC. Therefore goquic is actually broadly based on the Chromium QUIC implementation. The current status of goquic is highly experimental. The Go binding of the libquic library is in a pre-alpha state.

A disadvantage of goquic is the directory structure and the commentary. Neither the directory seems to follow a specific order nor is the code commentated. Moreover goquic has dependencies on libquic and Chromium QUIC. Therefore to modify goquic, a mixture of C++ and Go needs to be written.

quic-go [4] The quic-go implementation is in pure Go and not based on any other QUIC implementation. However quic-go is intended to be compatible with the current Google Chrome versions and the QUIC versions deployed on the Google servers.

1 The Chromium Projects: QUIC, a multiplexed stream transport over UDP URL *https://www.chromium.org/quic*

2 proto-quic URL *https://github.com/google/proto-quic*

3 QUIC support for Go URL *http://devsisters.github.io/goquic/*

4 A QUIC implementation in pure go URL *https://github.com/lucas-clemente/quic-go*

Quic-go has many advantages. The code is well documented and tested. The directory structure subdivides different components of QUIC (e.g., congestion control, ACK handling, HTTP over QUIC). After downloading the open-source code of quic-go the example server and client can be easily started.

We conclude that quic-go matches all requirements for our MPQUIC prototype. Compared to Chromium QUIC, it has fewer dependencies and deployment is easier. To obtain a fast available MPQUIC prototype, quic-go is the most reasonable existing QUIC implementation. We expect to be able to modify and deploy changes very easily.

5.1.1 Analysis of Multipath remnants in Chromium QUIC

The current version of the Chromium QUIC is not multipath capable. However QUIC was formerly intended to support multipathing in the future. Based on this intention some multipath supporting components have been implemented and can be found in the history of the chromium repository. Unfortunately the multipath approach was never accomplished and the intended use of the multipath components can only be assumed. Regardless of the use of quic-go for the MPQUIC extension, we analyze the remnants of first multipath compatible chromium QUIC components in the following.

Since January 2017 the developers of the Chromium project are actively removing the multipath components. The latest commit before the removal began was the 13th of January 2017. The corresponding Commit ID is `b25a2cd51e7758bf4c4f977cd01b17759e865bbf`. When inspecting this particular commit, a `Multipath Sent Packet Manager`, a `Multipath Receive Packet Manager` and `PathIDs` can be found. The `Multipath Sent Packet Manager` and the `Multipath Receive Packet Manager` manage per path available `Send Packet Managers` and `Receive Packet Managers` respectively. Based on the `PathIDs`, corresponding `Send-` and `Receive Packet Managers` are selected. These packet managers track sent and received QUIC packets of the corresponding QUIC connection. The `Multipath Sent Packet Manager` and `Multipath Receive Packet Manager` are optional components, only used when multipathing is enabled. To decide if multipath managers are used, a `Multipath Enabled` flag in the packet header is used. When `Multipath Enabled` is not set, the `Multipath Sent Packet Manager` and `Multipath Receive Packet Manager` are replaced by a single `Send Packet Manager` and `Receive Packet Manager` respectively.
Based on the source-code, we expect that a `Multipath Transmissions Map`, which is never used by any other class, was intended to keep track of packets that are sent on more than one path. When a packet retransmission occurs on a different path than the packet was originally sent, an entry is added to this map. Although the implementation of this map existed, it was never used.
Moreover to close paths, methods called `OnPathCloseFrame` and `SendPathCloseFrame` are implemented. Therefore we expect that QUIC designers intended to use an unspecified frame type `PathClose`. When inspecting the additional paths, we noticed that the initial packet number is always 1, confirming the design of MPQUIC (Section 4.3). Furthermore we notice that the multipathing does not effect the streams or the frames.

5.1.2 Analysis of quic-go

The implementation of MPQUIC is an extension of the open-source QUIC implementation quic-go. HTTP over QUIC, including the HTTP/2-QUIC Stream mapping, as described in Section 3.1.5, is included in quic-go to transmit responses and requests. Since quic-go is on active development the GitHub commit with *CommitID* `7a49b06c6c05a3c3927c7e6abde47121fe913a0e` is used for MPQUIC. This commit is the latest commit when starting the implementation of MPQUIC. The entire implementation and the analysis of quic-go is based on this commit
For analysis reasons we separate quic-go into components based on mechanisms and responsibilities.

These components can be considered individually. A overview of the components is given in the appendices Figure A.1
At first we consider the QUIC Server and QUIC Client. The Server and Client are responsible for listening on a specific UDP-Socket. Whenever data , e.g., QUIC packets, is read from the UDP-Socket, the ConnectionID is used for matching the packet with a QUIC connection. Based on the packet type further processing is triggered by the Server or the Client. When no connection has been established yet, Clients are responsible for sending the initial handshake and trigger the creation of a intern connection. On the other side, Servers receiving handshake packets are responsible for triggering the creation of intern connections.
A connection in quic-go is handled by a component called QUIC Session. In case intern connection creation is triggered, a Session is created and also handles connection establishment. At Session setup, a stream handler, sent- and receive packet handlers, flow controller, packet packer and unpacker, crypto components and a congestion controller is created. Additionally the Session handles the crypto stream for sharing cryptographically information. After the encryption status of a connection is forward-secured, a h2q (HTTP over QUIC) entity is responsible for handling header streams and serving HTTP content (application data). This includes writing response data on HTTP/QUIC Streams.
For sending this data, the Session first asks the congestion control if sending is allowed. Based on the congestion window and the bytes in flight, sending may be allowed or not. When sending is not allowed, the Session may send retransmissions but has to wait for sending fresh data until ACKs are received. When sending is allowed (i.e., the congestion window is not exhausted), the Session triggers the composing of the payload. Retransmission data, ACK frames and control frames are collected at first. Before collecting stream data, the connection and stream flow controllers is asked. When the flow controllers allow the sending of further stream data, a stream framer is used for framing the stream data based on the remaining payload size. Finally the payload is packet into Frame Packets, sealed and given to the UDP-Socket for sending. When sending, the sent packet handler and the congestion control is also notified.
Frame Packets received at Clients and Servers after connection establishment are matched to a corresponding Session which processes these packets by using the components created at setup. At first received packets are being unsealed and unpacked while a receive handler manages the creation and queuing of ACK frames. After unpacking the payload, the Session can handle the frames, obtained out of the payload, dependent on their type. ACK frames for example trigger specific actions at the sent packet handler and at the congestion control. Application data obtained from stream frames is composed to ordered byte streams and given to the application on top of QUIC, i.e., HTTP over QUIC.
According to the QUIC specification [25], Clients may send early data after sending the complete CHLO. Thus the first data in QUIC can be received by the server after 1.5 RTT's. In quic-go first data is sent after the SHLO is received by the client. This leads to a delay of 2.5 RTT's until the server receives the first bytes of application data.

5.2 Implementation of MPQUIC in quic-go

The implementation of MPQUIC is based on the existing open-source *quic-go* implementation. Figure 5.1 illustrates the major changes of quic-go for multipathing support. In quic-go, the core packet processing component is the QUIC-Session with a unique ConnectionID. Packets with the same ConnectionID pass the same handler instances and controllers.
Our previously introduced MPQUIC design has a per subflow packet number space and RTT-statistics, as well as a decoupled congestion control. Hence, MPQUIC requires certain components on a per subflow basis, i.e., the Sent and Received Packet Handler, Congestion Controller, RTT-Statistics and a UDP-Socket. The Session Multipath Manager handles subflow creation and keeps track of subflow specific components. Whenever a packet is processed by the QUIC-Session, the Session Multipath

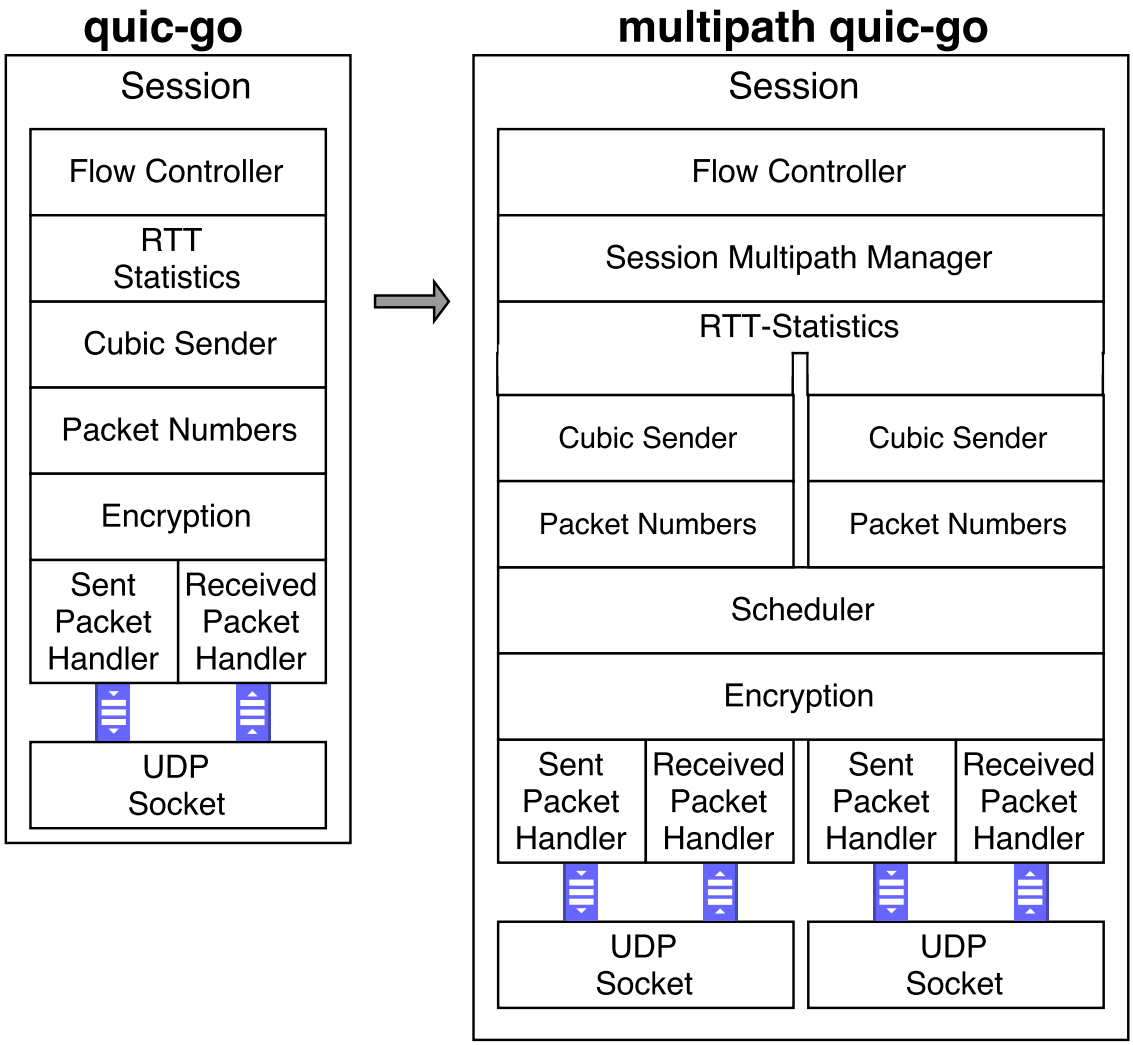


Figure 5.1.: The design of an existing QUIC implementation and its MPQUIC extension.The `Session Multipath Manager` keeps subflow specific components. The `Scheduler` decides on which subflow to send data.

`Manager` provides the subflow based components. Accordingly, in our MPQUIC implementation the subflow of a packet determines the instance of handlers and controllers which process a packet in addition to the `ConnectionID`. To decide on which subflow to send upcoming data a scheduler is implemented. The scheduler composes packets based on the available data per stream and chooses the according subflow for this packet.
Additionally, the ability for `Clients` and `Servers` to handle `Announcements` is implemented. Currently only `Clients` can send `Announcements` based on the detected network interfaces. When `Servers` send data or `Clients` receive data for the first time over a new subflow, the setup of additional subflow components is triggered at the `Session` and the `Session Multipath Manager`.

Initial Congestion Control Window

While implementing MPQUIC for quic-go we noticed that the initial congestion control window size is set to 32 packets. However the specification of QUIC [25] specifies the size of the initial congestion window is the same as in TCP Cubic (i.e., 10 packets). The effect is that more data may be sent initially duo to the higher window. Compared to TCP or other protocols, that use an initial congestion window of 10 packets, quic-go is not fair and not comparable.
For flexibility and later evaluations, a command-line command (i.e., "$-c$") for setting the initial conges-

tion window is added. The parameter passed via "−c" is an unsigned integer determining the amount of packets of the initial congestion window (e.g., $./server \quad -c = 10$). Additionally the default value of the initial congestion window, when "−c" is not used, is adjusted to 10 packets.

Accept STK

QUIC servers may only be willing to send its configuration if a valid source-address-token (STK) is provided (Section 3.5). This willingness influences the amount of RTTs required for the initial handshake. In quic-go the default method for accepting the STK requires a valid STK before allowing to send its configuration. This requires an additional RTT, compared to the handshake described in [36] and depicted in Figure 3.5. Especially critical is this for later evaluations when comparing quic-go with TCP or MPTCP. Therefore the default method for accepting the STK is adjusted in quic-go and multipath quic-go. The server is changed to send all required information for a complete CHLO in its very first REJ, even if no STK is provided by the client in the inchoate CHLO. Note that this change still follows the traditional QUIC specification.

Adjustments for Evaluations

Evaluations of MPQUIC include a comparison against HTTP/2 over TLS/TCP and TLS/MPTCP. However the native quic-go server implementation only provides HTTP 1.1 over TCP. Accordingly the server implementation of quic-go is adjusted to use HTTP/2 over TLS/TCP (HTTPS). Moreover a TCP client in quic-go is implemented.
In later experiments webpages are requested and loaded using HTTP/2. The task is to request the main file of the webpage first and successive request all additional required files (i.e., images, scripts, style sheets) for that webpage. For this purpose a simple HTML scraper[5] is implemented and used for the QUIC and TCP client. The scraper searches for certain html tags (i.e., *link*, *script*, *img*) and extracts the corresponding URL out of the attribute values (i.e., *href*, *src*). Next the clients request the extracted URLs.

5.3 Plug and Play Scheduler

Despite the implementation of the Lowest-RTT scheduler, it is desired to have the possibility to add and select additional schedulers. For this purpose a `schedulerFunctionLambda` is declared. The `schedulerFunctionLambda` parameters are `SchedulerAuxiliaryData`, that provide recent path specific information, and a general `Scheduler` with path overreaching information and methods. Using this parameters, the scheduler function has access to ACK frames, retransmission data, data per stream (including `StreamID`, `Stream Priority` and data length), RTT-statistics for ever subflow and much more information, necessary for fine-grained scheduling. An overview of the `schedulerFunctionLambda` and corresponding type structs is provided in Listing 5.1.
Based on a command-line command (−*sc*) individual defined scheduling functions, that follow the declaration of the `schedulerFunctionLambda`, are selected after compilation of MPQUIC. For selecting the default scheduler the string "mrtt" (MinimumRoundTripTime) is passed via −*sc* in the command line:

$$./server \quad -sc = "mrtt"$$

For selecting the included RoundRobin scheduler the string "rr" needs to be passed. If no scheduler is specified via command-line (−*sc*), the Lowest-RTT scheduler is selected by default.

[5] https://schier.co/blog/2015/04/26/a-simple-web-scraper-in-go.html

```
type schedulerFunctionLambda func(SchedulerAuxiliaryData,
                        *Scheduler)
                        returns (map[PathID][]frames.Frame, error)

type SchedulerAuxiliaryData struct {
        AllowedPaths              []PathID
        PathsWithRetransmission   []PathID
        AckFramesMap              map[PathID]*frames.AckFrame
        StopWaitingFramesMap      map[PathID]*frames.StopWaitingFrame
        PublicHeadersLength       map[PathID]ByteCount
        ...
 }

type Scheduler struct {
        dataToSend                 []SchedulerSendData
        controlFrames              []frames.Frame
        streamFramer               *streamFramer
        streamsMap                 *streamsMap
        rttStatsManager            *RTTStatsManager
        ...
  }
type SchedulerSendData struct {
        StreamId                  StreamID
        Priority                  PriorityParam
        DataByteLength            ByteCount
}
```

Listing 5.1: The declaration of the `schedulerFunctionLambda` and corresponding type structs.

A similar concept exists for the stream scheduler to enable future stream to subflow scheduling optimization. The `streamSchedulerFunctionLambda`'s parameters are a `streamsMap`, which serves open streams, and `streamLambda` callback (Listing 5.2). The `streamLambda` needs to be defined in the `schedulerFunctionLambda`, to extract data out of a stream. The command-line command for selection a stream scheduler is "*–ssc*". The default implementation is a RoundRobin stream scheduler. Additional several experimental HTTP/2 priority based schedulers are included in MPQUIC but are not recommended for current use.

```
type streamSchedulerFunctionLambda func(*streamsMap,
                        streamLambda)
                        returns error
```

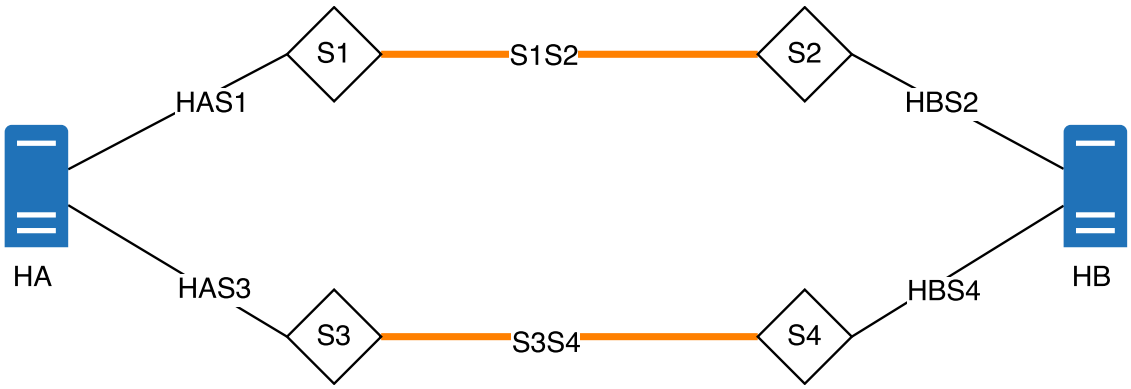
Listing 5.2: The declaration of the `streamSchedulerFunctionLambda`.

6 Analysis

The design of the multipathing extension for QUIC called for many decisions. A prototype of MPQUIC is implemented in quic-go. When designing and implementing such an extension, various aspects need to be analyses and tested for plausibility before a large evaluation and comparison with other protocols is reasonable. This chapter provides plausibility tests of MPQUIC and reveals findings during the analysis.

6.1 General Setup

(a) General setup for a single path netwotk.



(b) General setup for a multi-path netwotk.

Figure 6.1.: Illustration of the general test setup. The identifiers H are hosts (e.g., HA:Host A), S are switches (e.g., S1:Switch 1) and links are named according to the network device they connect (e.g., HAS1 connects Host A with Switch 1).
The path limitations are adjusted by changing the parameters of links between switches (the orange links).

Mininet[1] is used to setup simulated networks that match defined experiments. With Mininet it is possible to run a real network stack in user defined topologies. Once the networks are set up, files are requested by clients using HTTP/2. For our experiments, paths consist of two endpoints (hosts), two switches and three links connecting the hosts and the switches. One host has exactly one network interface for each path. This network interface is connected to one switch. A switch is connected to one host and one other switch. The general setup, used for our experiments, is illustrated in Figure 6.1. For the individual setups, the bandwidth bw and the delay d of the links between switches (i.e., Figure 6.1, orange links) are varied. The links between hosts and switches have a bandwidth of $bw = 100$ Mbps and a delay of $d = 1$ms and remain untouched for all experiments.

[1] Mininet URL *http://mininet.org/*

6.2 Plausibility Analysis

For verification of the correctness of MPQUIC and its implementation, testing the behavior is necessary. Since there is no other QUIC Multipath implementation, there is no previous work to compare our results with. However with simple test setups, where we are able to state result expectations, it is possible to test the plausibility of the MPQUIC design and implementation. The test setups are designed to verify simple behavior that should be fulfilled by MPQUIC in theory.
At first, simple setups, including expected results that aim to verify a certain behavior, are created. The test setups use either one or two paths and differ in the link parameters (i.e., bandwidth and delay) of those paths. Based on the expectation and the obtained test result, a statement about MPQUIC's plausibility is made. In each setup, traditional single path QUIC is compared with MPQUIC. Additionally MPQUIC is tested with the Lowest-RTT-Subflow first (MinRTT) and the RoundRobin scheduler. The parameter settings for the switch-to-switch links are listen in Table 6.1.
The path used for connection establishment is denoted as *first path* in the following. In the single path setup this is the only available path. The additional path, which is announced after the connection establishment, is denoted as *second path*. The requested file size for the plausibility setups is set to 1 MB. This file size is chosen to lower the influence of the time required for connection and path establishment.

Table 6.1.: The bottleneck link parameters of the test setups. The red and blue numbers highlight derivations from the default parameters.

Setup		1	2	3
First Path	bw	10	10	10
	d	20	20	20
Second Path	bw	10	5	50
	d	20	100	10

Setup 1: Identical Paths

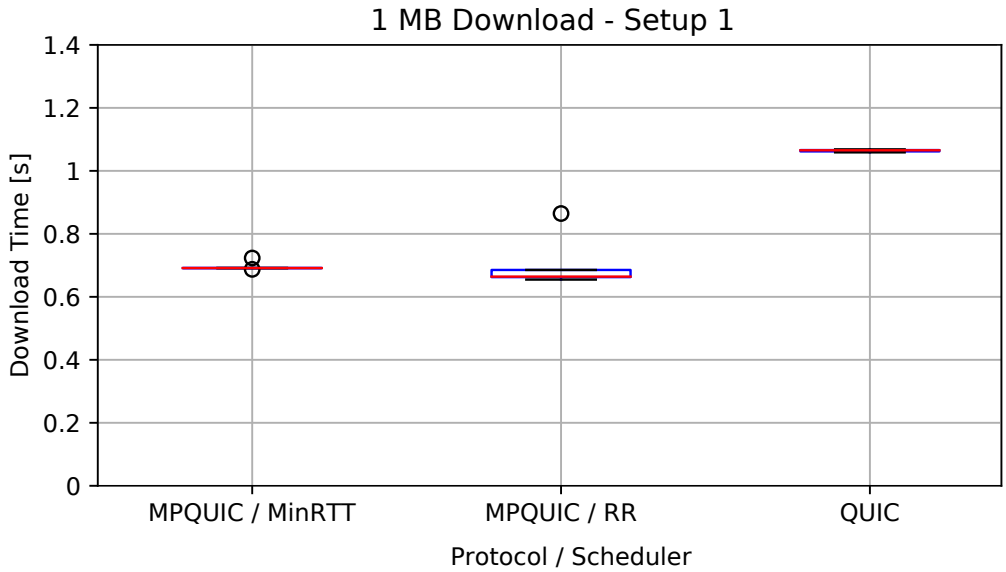


Figure 6.2.: The download time for MPQUIC using the MinRTT and RoundRobin scheduler in comparison with traditional QUIC. The available paths have a bandwidth of 10 Mbps and a RTT of 44ms. MPQUIC reduces the download time by more than 30%.

The most simple test case is a multipath setup with one additional path that has identical characteristics as the initial path. Therefore both paths have a bandwidth of 10 Mbps and a delay of 20ms in setup 1.
The objective of this setup with two identical paths is to verify that the download time decreases. Even simple scheduling techniques should be able to achieve high speedups.
It is expected that both, the MinRTT and the RoundRobin scheduler, are able to achieve a speedup of 25%-50%. This is duo to the equal bandwidth and delay of the both paths. Simply spoken the bandwidth is added up while the delay remains the same. This setup has a total bandwidth of 20 Mbps by a constant delay of 10ms.
The results of the first setup are visualized in Figure 6.2. Traditional QUIC that uses only a single path requires 1.05s for downloading the 1 MB file. MPQUIC achieves a mean download time of less than 0.7s using the RoundRobin or the MinRTT scheduler. This corresponds to a speedup of more than 30% compared with traditional QUIC. It is verified that MPQUIC is able to decrease the download time in a environment with two identical paths.

Setup 2: Limit the second path

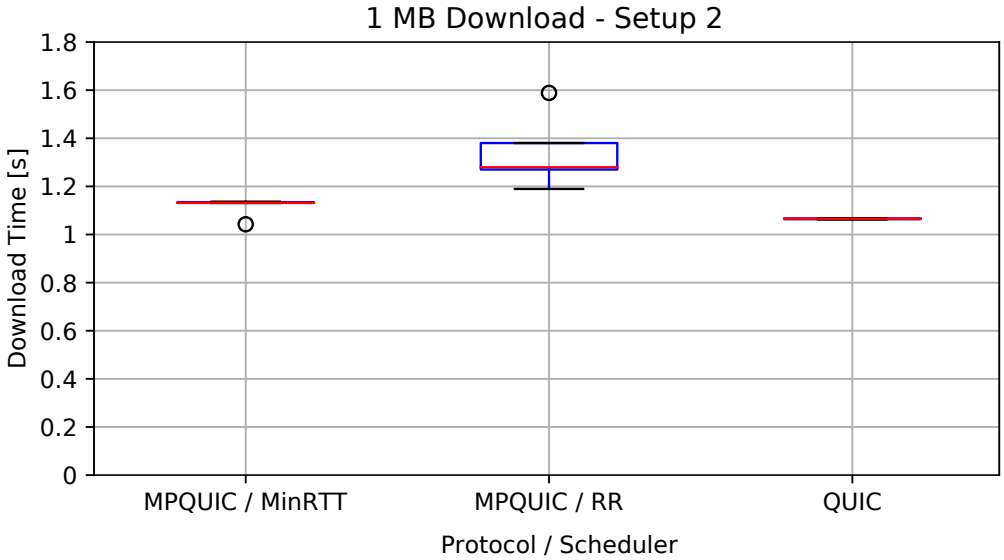


Figure 6.3.: The download time for MPQUIC using the MinRTT and RoundRobin scheduler in comparison with traditional QUIC. The initial path has a bottleneck bandwidth of 10 Mbps and a delay of 20 ms. The additional path, used by MPQUIC, has a bottleneck bandwidth of 5 Mbps and a RTT of 204ms.

Setup 2 is to verify that an additional path with limited bandwidth and higher delays, compared to the first path, results in higher download time if an inappropriate scheduling technique is used. Therefore setup 2 enforces poor multipath results by setting the bandwidth of the second path to only 1 Mbps and increases the delay to 100ms. Simple scheduling techniques that use both paths equally often will suffer from this limitations.
It is expected that the RoundRobin scheduler increases the download time compared to traditional QUIC because the equal use of both paths. The download time of MPQUIC using MinRTT is expected to be lower than MPQUIC with the RoundRobin scheduler. The download time of MPQUIC using MinRTT compared to traditional QUIC is uncertain. When the path with the lowest RTT is congestion blocked, MinRTT schedulers will utilize the other path. Since the file size is 1 MB the first path will eventually be blocked by the congestion window and the MinRTT will select the second path at some point. For the worst case (i.e., the last packet is scheduled on the second path) the download time should only increase by 160ms (i.e., difference of the paths RTTs).

Figure 6.3 confirms the expectation. MPQUIC with the RoundRobin scheduler utilizes the limited second path excessive. The download time compared to QUIC is increased by 20%. MPQUIC with the MinRTT scheduler is slightly worse than traditional QUIC. The reason is the high delay of the second path, which is also utilized with the MinRTT scheduler. However the MinRTT scheduler's performance is better compared to RoundRobin since the lower RTT is recognized and the second path is only utilized when the initial path is congested.

Setup 3: Boost the second path

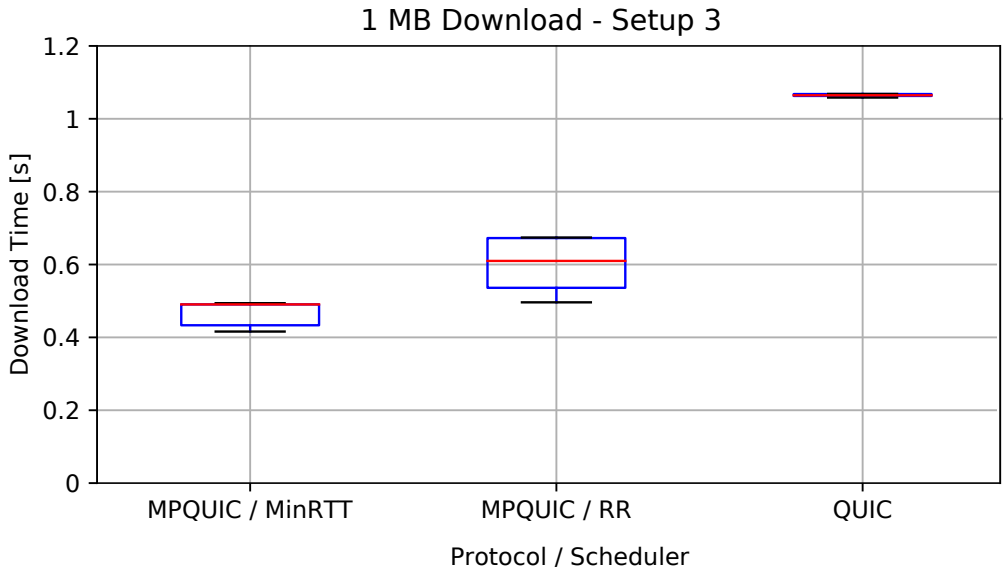


Figure 6.4.: The download time for MPQUIC using the MinRTT and RoundRobin scheduler in comparison with traditional QUIC. The initial path has a bottleneck bandwidth of 10 Mbps and a delay of 20 ms. The additional path, used by MPQUIC, has a bottleneck bandwidth of 50 Mbps and a RTT of 24ms.

The presence of an additional path with higher bandwidth and lower delay compared to the initial path allows an multipath protocol to significantly reduce the download time. Schedulers recognizing the low delay of the second path will benefit from the huge bandwidth increase (i.e., more then two times the bandwidth of the initial path). Therefore setup 3 provides a second path with a bandwidth of 50 Mbps and a delay of 10ms.
The goal of setup 3 is to verify that the MinRTT recognizes the lower RTT path and utilizes it more than the RoundRobin scheduler. It is expected that the MPQUIC using MinRTT has a lower download time compared to MPQUIC using RoundRobin duo to increased use of the path with more bandwidth.
The lower download time of MPQUIC using MinRTT compared to MPQUIC using RoundRobin is illustrated in Figure 6.4. The experiments verify that the MinRTT scheduler recognizes a lower RTT path and uses it more often compared to RoundRobin.

6.3 Strive for the theoretical maximum

Multipathing protocols, in comparison with single path protocols, that use one additional path with the same characteristics as the initial path (i.e., homogeneous paths) achieve speedups in download time with the theoretical maximum of factor 2. Nevertheless we show that this speedup is hard to realize. As we will see later, with increasing file size in homogeneous environments, multipath protocol achieve speedups closer to the maximum of factor 2. In the following we argue why the maximum speedup is hard to realize and analyze an example. For better illustration, small files are requested in this section.

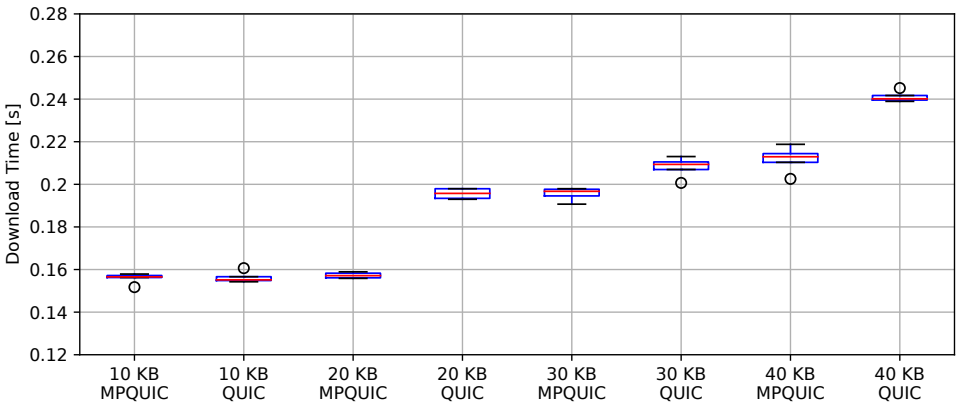


Figure 6.5.: The download time for MPQUIC using the MinRTT scheduler in comparison with traditional QUIC. The available paths have a bandwidth of 10 Mbps and a RTT of 44ms.

Figure 6.5 shows first comparisons of MPQUIC , using the MinRTT scheduler, and QUIC. Several files of different sizes were requested and loaded. For the selected files the theoretical maximum speedup is never achieved duo to the subflow establishment that occurs after the initial handshake. This means that the handshake adds a constant delay that occurs in QUIC as well as in MPQUIC and can not be decreased with multiple paths. With increasing file size and thus increasing total download time, the impact of this constant delay is attenuated.
Another phenomena, revealed in Figure 6.5, is that MPQUIC is as good as QUIC when requesting a 10 KB file However the speedup increases significantly for a 20 KB file request but decreased again for a 30 KB file. The reason for this phenomena is the size and the growth of the congestion window. The default initial congestion window size equals 10 packets, each carrying 1350 bytes in the payload (Section 5.2). Hence QUIC is allowed to send 13.5 KB while MPQUIC may send up to 27 KB of application data in the first data transmission round after the handshake. QUIC requires one RTT for sending 10 KB and two RTTs for sending 20 KB. MPQUIC has one congestion window for each subflow enabling to send even the 20 KB file in the first RTT after the handshake.

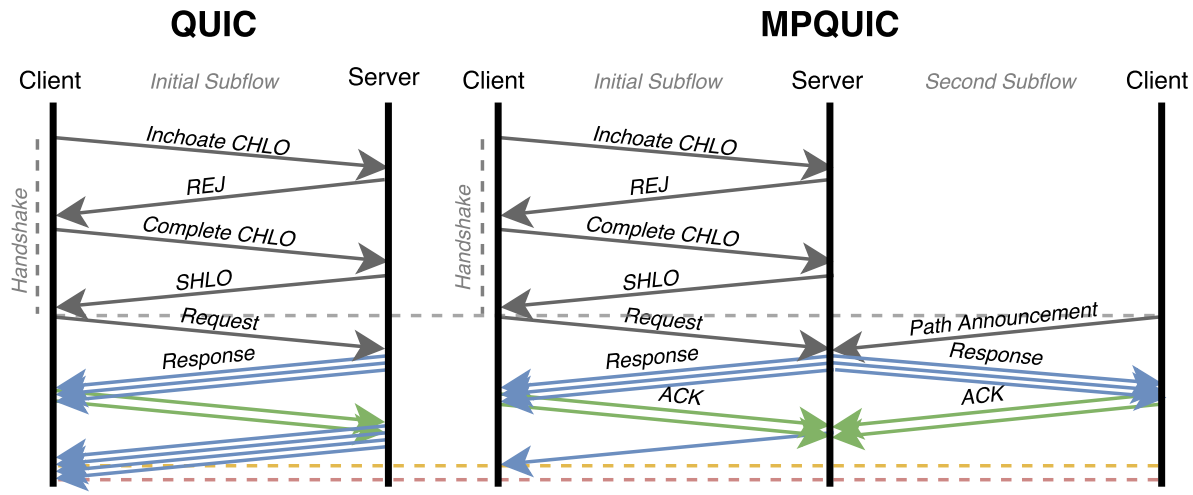


Figure 6.6.: Timelines for requesting 30 KB file with QUIC (left) and MPQUIC (right). The 30 KB do not fit into two initial congestion windows of MPQUIC. Thus MPQUIC requires two rounds for sending 30 KB. QUIC needs as well only two RTTs. The growth of the congestion window lead to a sufficient large window to send 30 kb in two rounds.

For further illustration the timeline of a 30 KB file request is depicted in Figure 6.6. QUIC sends 13.5 KB and MPQUIC sends 27.5 KB of application data in the first RTT duo to the initial congestion windows. Both protocol are not able to sent 30 KB in one round. When the first ACKs are received, MPQUIC can send the remaining 2.5 KB in the second RTT. At the first sight one may mistakenly conclude that single path QUIC can not send the remaining 16.5 KB in its second RTT. However duo to the congestion window growth, more than 10 packets can be sent in QUIC's second round. Thus both single and multipath QUIC require two rounds of sending data for a 30 Kb file.

6.4 Stream limitations: HTTP Write Buffers

For downloading files over HTTP in quic-go, clients send HTTP request to servers. The HTTP over QUIC entity of a server is responsible for processing the HTTP request and serving the file. After interpreting the request, the HTTP entity loads the requested file from the file system. Next the content of the file is written into buffers. These buffers have a size of only 32 KB. When data larger 32 KB needs to be written, the bytes slice is split to match the buffer size. After splitting and filling a buffer, a `ResponseWriter` receives a single buffer per call and writes the data of the filled buffer onto an QUIC-Stream. Only after the QUIC-Stream data has been read by a framer, the next buffer is written on the QUIC-Stream. This process of reading files, filling the buffers and writing the stream data is done in parallel for all received requests.
Duo to the limited buffer size of 32 KB, QUIC-Streams are not aware of the current data size to send over a certain stream. This information is especially important for stream schedulers that want to schedule urgent data first. When only 32 KB of data are written at once, a stream scheduler does not know that there is more urgent data and will only schedule 32 KB of the urgent data at once, and may fill a congestion window with data of other, less urgent streams.

7 Evaluation

As QUIC originates in HTTP/2, we use HTTP/2 file downloads of varying sizes for the evaluation. Moreover since QUIC provides always full encryption of regular packets, we use HTTPS and thus TLS for TCP and MPTCP.
First the speedup when using multiple paths is evaluated. For this purpose the download time of HTTP/2 file requests over traditional QUIC and MPQUIC is compared. To obtain results for different network environments, homogeneous networks and heterogeneous networks are set up using Mininet. Homogeneous networks are characterized through paths with the same bandwidth and RTTs. Heterogeneous networks have paths with different bandwidth and RTTs. After the comparison with QUIC, MPQUIC is compared with the state-of-the-art transport protocols TCP and MPTCP. Again homogeneous networks and heterogeneous networks are set up. For the evaluation the setup described in Section 6.1, is used. If not stated otherwise, the default Lowest-RTT-First Scheduler (MinRTT) is used.

7.1 Speedup of Multipath QUIC

The speedup of MPQUIC corresponds to the decrease of the download time compared to single path QUIC. Decreasing the download time T_0 to a download time of $T_1 = \frac{1}{X} \times T_0$ equals a speedup of factor X. This section evaluates the speedup achieved with MPQUIC.

7.1.1 Homogeneous Networks

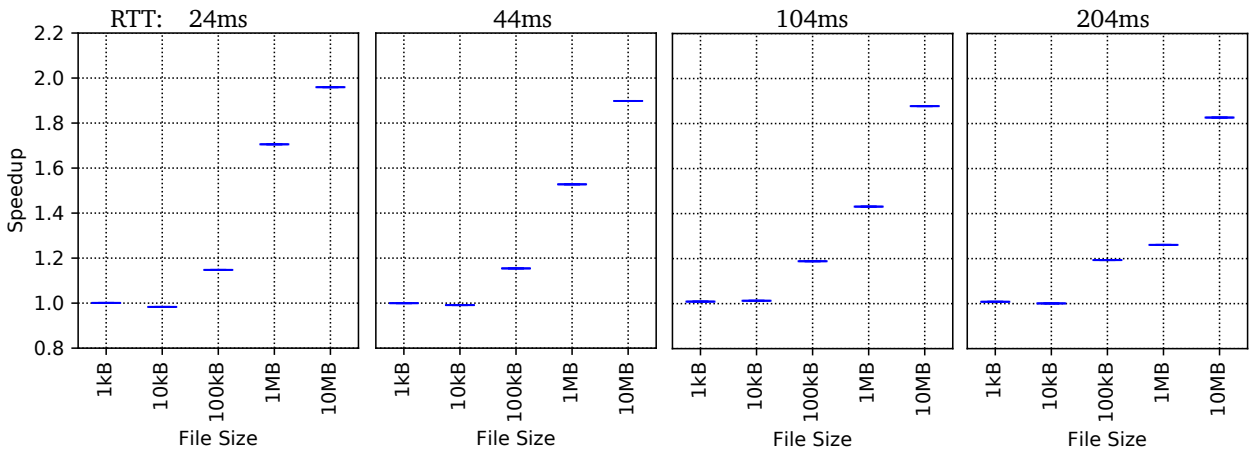


Figure 7.1.: Speedup of MPQUIC using the MinRTT scheduler. The homogeneous paths have a bandwidth of 10 Mbps. From left to right, the RTTs of both paths increase.

In a homogeneous network, all paths have the same latency and capacity. For the experiments we obtain path parameters out of a set of bandwidths $bw_s = [1, 10, 100]$Mbps, and $RTT_s = [24, 44, 104, 204]$ms and use the MinRTT and RoundRobin schedulers in MPQUIC. All possible $(Scheduler \times bw \times RTT)$ combinations are evaluated. Figure 7.1 visualized the speedup of MPQUIC with the MinRTT scheduler for a fixed bandwidth of 10 Mbps and various delays. The figures show that the speedup increases with the file size. Small files do not benefit from multipathing. However we note that we are as good as traditional single path QUIC. These small downloads are not throughput limited by the congestion window, but suffer from the constant delay of the connection establishment and the HTTP/2 request. For files of 100 KB, however, MPQUIC already provides a significant speedup. For larger files, the speedup of the

download time nearly reaches the theoretical maximum of a factor of 2.
A similar behavior is observed when evaluating different $Scheduler \times bw \times RTT$ combinations. For a detailed visualization see Section B.1 of the appendices. In the homogeneous environment the RoundRobin scheduler achieves similar speedups as the MinRTT scheduler. With increasing bandwidth the speedup decrease slightly duo to overall download time reduction and thus the impact of the connection establishment delay increases.

7.1.2 Heterogeneous Networks

Based on the experiments with homogeneous paths, we now analyze the impact of path heterogeneity in terms of bandwidth and RTTs. Here, we consider a scenario with 2 paths. We divide this evaluation in two parts. First we vary the bandwidth of the additional path, next the RTT is varied.

Bandwidth Heterogeneity

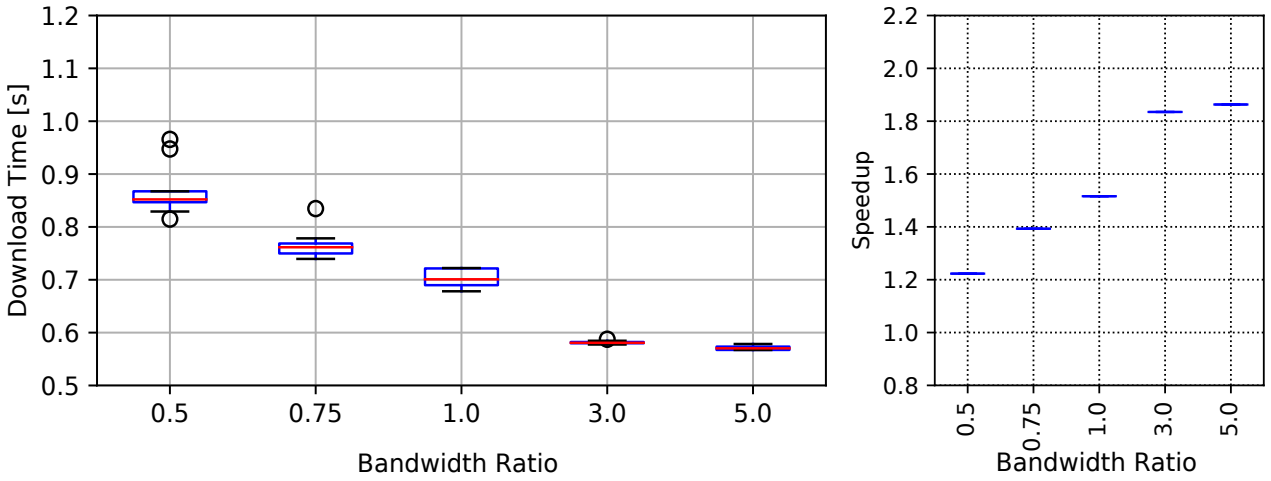


Figure 7.2.: Impact of different bandwidth ratios. The requested file has a size of 1 MB. The initial subflow has a bandwidth of 10 Mbps. All subflows have a RTT of 44ms.

We set the bandwidth of the first path to $B_1 = 10$ Mbps and vary the bandwidth ratio of the first and the second path B_2/B_1 for a 1 MB file and homogeneous round-trip times. Figure 7.2 illustrated the download time of MPQUIC and the speedup compared to traditional QUIC for various bandwidth ratios. The measurements show that MPQUIC benefits from additional paths even if these paths only offer a low bandwidth. In this experiment, adding a low bandwidth subflow with half the capacity of the first subflow gives MPQUIC a speedup of factor 1.2. When increasing the bandwidth of the additional path, MPQUIC is able to further decrease the download time. Having a path with five times the capacity leads to a speedup of over 1.8. However with the increased capacity, the theoretical maximum speedup also increased (i.e., factor 2 for ratio 1.0, factor 4 for ratio 3.0, factor 6 for ratio 5.0). Furthermore we note that for a bandwidth ratio of 3.0 and 5.0 the download time is nearly the same. We assume that the reason for the limited speedup is the file size in combination with the growth of the congestion window. The maximum possible congestion window size may not be reached when loading a 1 MB file and thus ratios of 3.0 and 5.0 utilized the same path capacities.
Figure 7.3 shows the download time and speedup when requesting a 100 MB file with MPQUIC. We observe that the speedup further increases and almost reaches it theoretical maximum for all tested ratios. MPQUIC almost achieves the maximum possible speedup, when the size of the requested file is large enough duo to the full utilization of the available bandwidth.

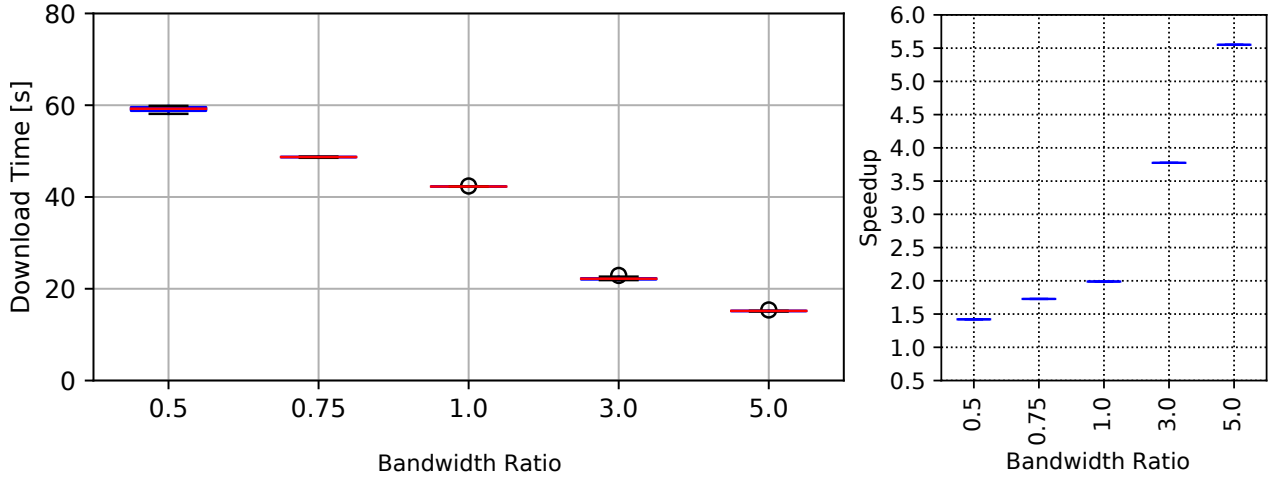


Figure 7.3.: Impact of different bandwidth ratios. The requested file has a size of 100 MB. The initial subflow has a bandwidth of 10 Mbps. All subflows have a RTT of 44ms.

Round-Trip Time Heterogeneity

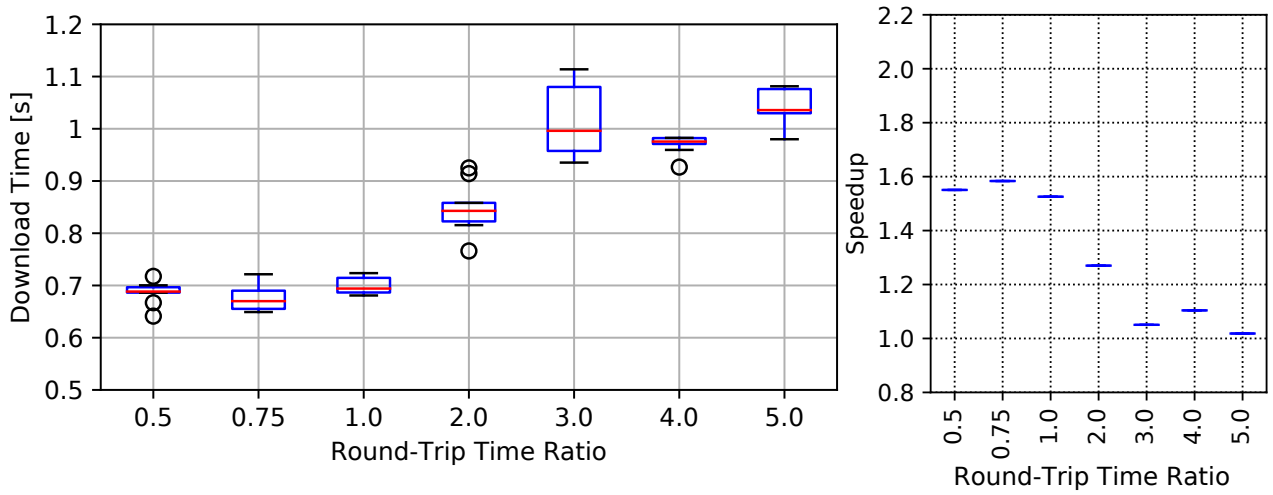


Figure 7.4.: Impact of different RTT ratios. The requested file has a size of 1 MB. All subflows have a bandwidth of 10 Mbps. The initial subflow has a RTT of 44ms.

Here we fix the RTT of the first path $R_1 = 44$ ms and vary the bottleneck delay D_{2b} of the second path. The resulting RTT ratio is $(2 \times D_{2b})/(R_1 - 4ms)$. The additional 4ms result from delays at links apart from the bottleneck. As already described Figure 6.1, only the bottleneck links are varied.
Figure 7.4 visualize the download time and speedup of MPQUIC, requesting a 1 MB file. The download time increases with the RTT of the additional path. Simultaneously the speedup decreases. While RTT ratios of 1.0 or lower cause speedups of almost 1.6 with MPQUIC, higher RTT ratios lead to nearly no speedup. Still MPQUIC always decreases the download time compared to QUIC.
Things look slightly different in Figure 7.5, where a 100 KB file is requested. The download time for a RTT ratio of 3.0 substantial increases and thus QUIC outperforms MPQUIC. The reason for the download time increase may be a combination of congestion control and bad scheduling. Every round a certain amount of data, based on the current congestion window, is sent. The MinRTT scheduler uses even poor performing subflows, when the congestion window of the lowest RTT path is exhausted. When data is sent on the subflow with the higher RTT in the last round of sending data, the download time increases, based on the different RTT values. In our example with a RTT ratio of 3.0, $R_1 = 44$ms and $R_2 = 124$ms,

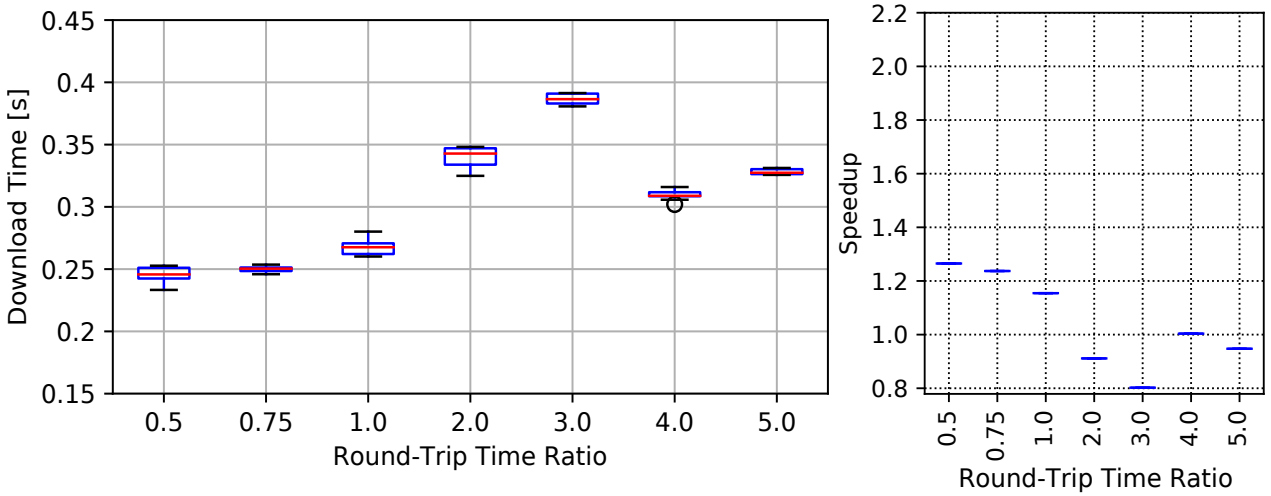


Figure 7.5.: Impact of different RTT ratios. The requested file has a size of 100 KB. All subflows have a bandwidth of 10 Mbps. The initial subflow has a RTT of 44ms.

packets sent on the first path are received by the client after approximately 22 ms. Packets sent on the second path with a higher RTT are received after 62ms. Thus bad scheduling in the last sending round increases download time [19].
On the contrary, Figure 7.6 shows the download time and corresponding speedup of MPQUIC when requesting a 100 MB file. Here the maximum possible speedup of factor 2 (i.e., both subflows have the same bandwidth) is almost reached even for a RTT ratio of 5.0. The impact of the additional delay, induced by using the high RTT subflow in the last sending round, is very low compared to the overall download time. This strengthens the argument, that bad scheduling in the last sending round is responsible for the slow down when requesting small files.

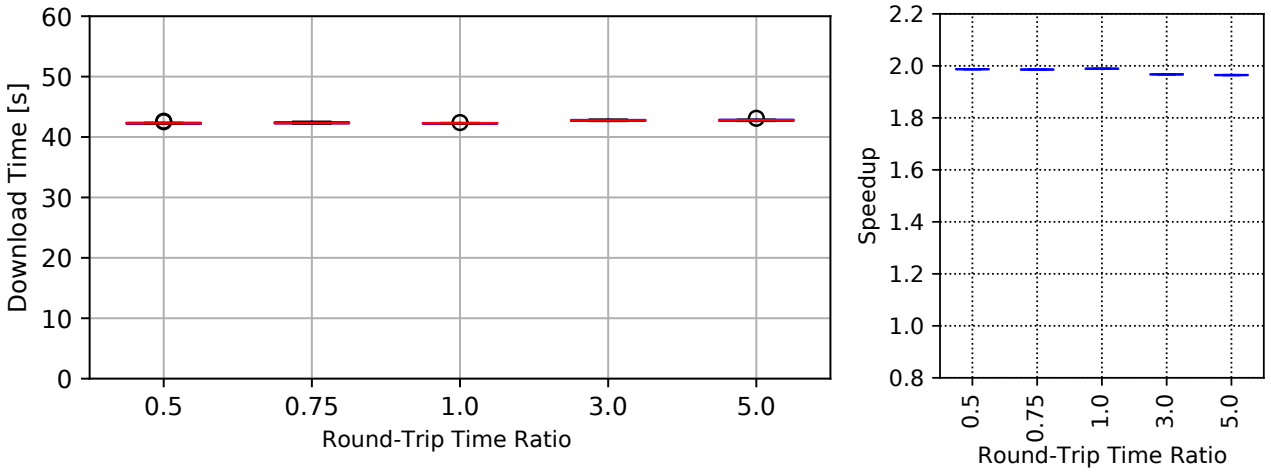


Figure 7.6.: Impact of different RTT ratios. The requested file has a size of 100 MB. All subflows have a bandwidth of 10 Mbps. The initial subflow has a RTT of 44ms.

7.1.3 Increasing the number of Subflows

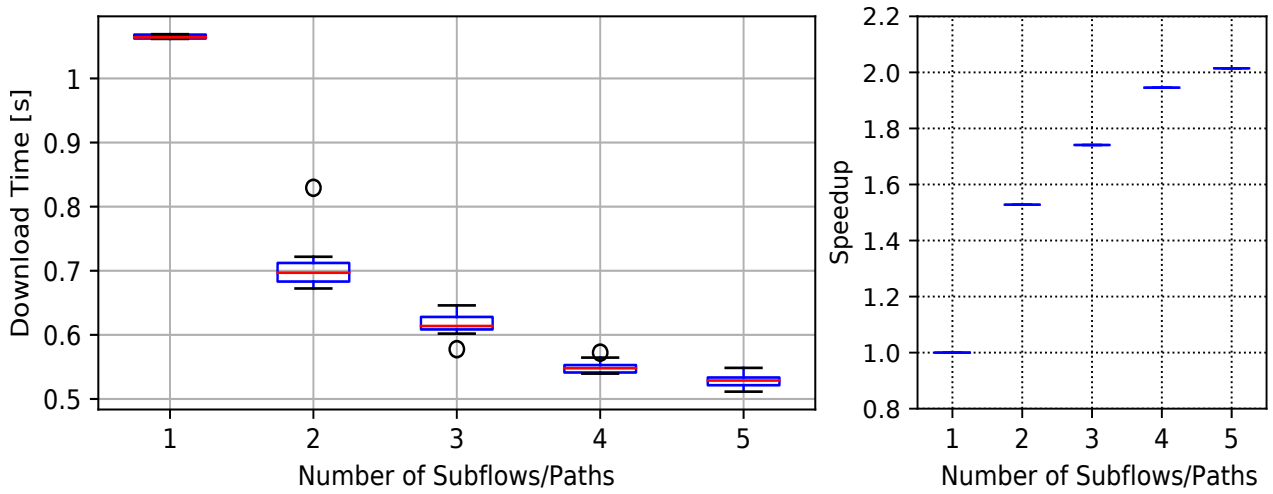


Figure 7.7.: Impact of an increasing number of subflows respectively paths. The requested file has a size of 1 MB. Each subflow has a bandwidth of 10 Mbps and a RTT of 44ms.

With an increasing number of subflows the available capacity increases. Thus a multipathing protocol may utilize more bandwidth and further decreases the download time. In a setup with homogeneous paths, the theoretical maximum speedup equals the number of paths. We now evaluate the download time of MPQUIC for different number of subflows.
A 1 MB file is requested in a homogeneous network with a bandwidth of 10 Mbps and RTT of 44ms each path. The result is shown in Figure 7.7. As the handshake and the HTTP request require a constant time, the speedup when using 5 homogeneous subflows is only of factor 2. The impact of this constant delay usually decreases when loading larger files, as the total download time increases.

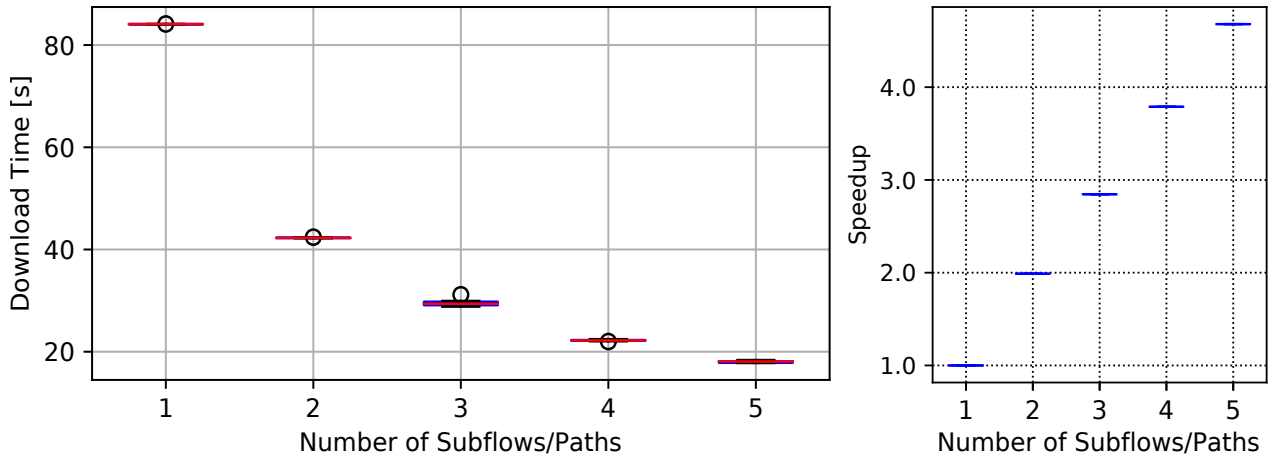


Figure 7.8.: Impact of an increasing number of subflows respectively paths. The requested file has a size of 100 MB. Each subflow has a bandwidth of 10 Mbps and a RTT of 44ms.

In Figure 7.8 a 100 MB file is requested. We see that the achieved speedup when using 5 homogeneous subflows is greater then factor 4. The download time is reduced from over 80s to less than 20s when using 5 subflows instead of 1. The theoretical maximum speedup of 5 is almost achieved. The impact of the constant time required for the connection establishment and the HTTP request decreases when the overall download time increases.

7.1.4 Conclusion

MPQUIC benefits from additional paths and is able to reduce the download time. The achieved speedup depends on the network characteristics and on the traffic pattern, e.g., the size of the requested file. In homogeneous networks, the theoretical maximum speedup is almost achieved by MPQUIC, when loading a sufficient large file. The delay induced by the connection establishment has significant impact on small files but is insignificant for larger files. A slowdown occurs when loading small files using an additional path with a substantial higher delay. Future work must focus on this slowdown by optimizing the scheduler. We suggest to avoid using subflows with a substantial higher delay in the last round of sending data.

7.2 Comparison of MPQUIC, MPTCP, QUIC and TCP

As we envision QUIC to become a universal transport protocol, we compare MPQUIC with the established state-of-the-art protocols TCP and MPTCP. For the MPTCP evaluations, we relied on the publicly available MPTCP Linux Kernel implementation [43]. In previous evaluations we showed the achievable speedup of multipath QUIC and the behavior of the download time for file requests of different sizes. The focus in the next evaluation is the behavior of MPQUIC compared to MPTCP. Moreover the behavior change between QUIC an MPQUIC is compared with the behavior change between TCP and MPTCP.

7.2.1 Homogeneous Networks

To analyze the general behavior of MPQUIC, we first measure the download time in homogeneous two-path networks. The measured download time for a 1 MB file is depicted in Figure 7.9. We notice that both multipath protocols achieve lower download times than their single path variants. Further the download time of QUIC and TCP, and the download time for MPQUIC and MPTCP are roughly the same. Thus the speedup of MPQUIC over QUIC is comparable with the speedup MPTCP over TCP.

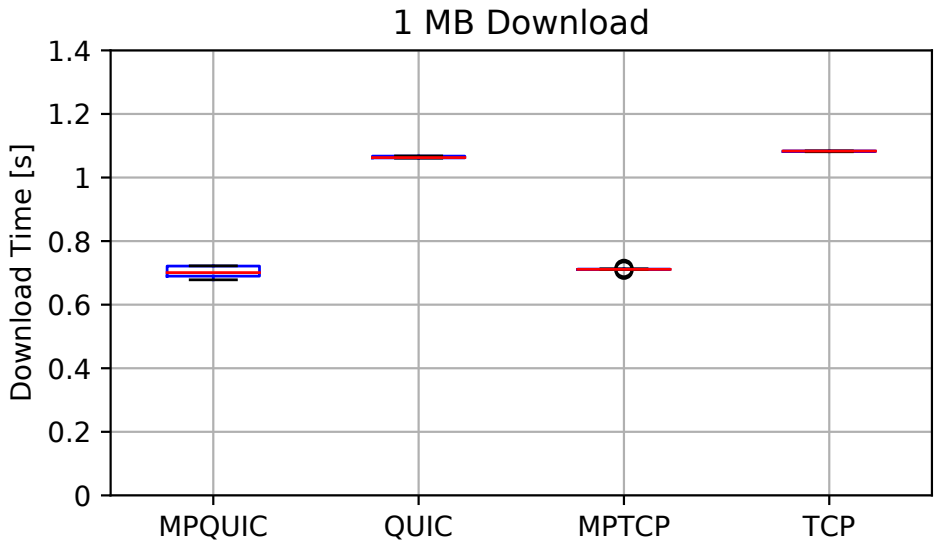


Figure 7.9.: Comparison of MPQUIC, QUIC, MPTCP and TCP for HTTP/2 file downloads. The bandwidth of each subflow is 10 Mbps. The RTT is 44ms.

The download time when requesting files of size 1 KB, 10 KB and 100 KB are shown in the appendices Section B.2.1. With decreasing file sizes, the impact of the constant time, required for connection and

subflow establishment, increases. The speedup for both multipath protocols, MPQUIC and MPTCP, decreases accordingly. However in an direct comparison of MPQUIC and MPTCP, in homogeneous networks MPQUIC achieves slightly lower download times.

Loading small files

We already showed that MPQUIC achieves a speedup near the theoretical maximum for larger files. In this section we want to evaluate and show the general behavior of MPQUIC after connection establishment (i.e., when requesting very small files). We also compare MPQUIC's behavior with that of MPTCP, TCP and QUIC. At first, files of size between 1 KB and 9 KB are loaded. Figure 7.10 visualizes the required download time of the four protocols. On one side we observe that the TCP based request have higher download times than QUIC based requests. QUIC combines the transport and the security handshake, while TCP requires an additional TLS handshake after connection establishment. Thus TCP has in general a higher handshake latency than QUIC [36]. On the other side, there is no significant difference between a request of 1 KB or 9 KB with respect to the individual protocols. Only single path TCP shows an increase in the download time for files of 8 KB and 9 KB. We conclude that MPQUIC's, MPTCP's and QUIC's initial congestion window is large enough to send 9 KB. TCP's initial window size must be between 7 KB and 8 KB.

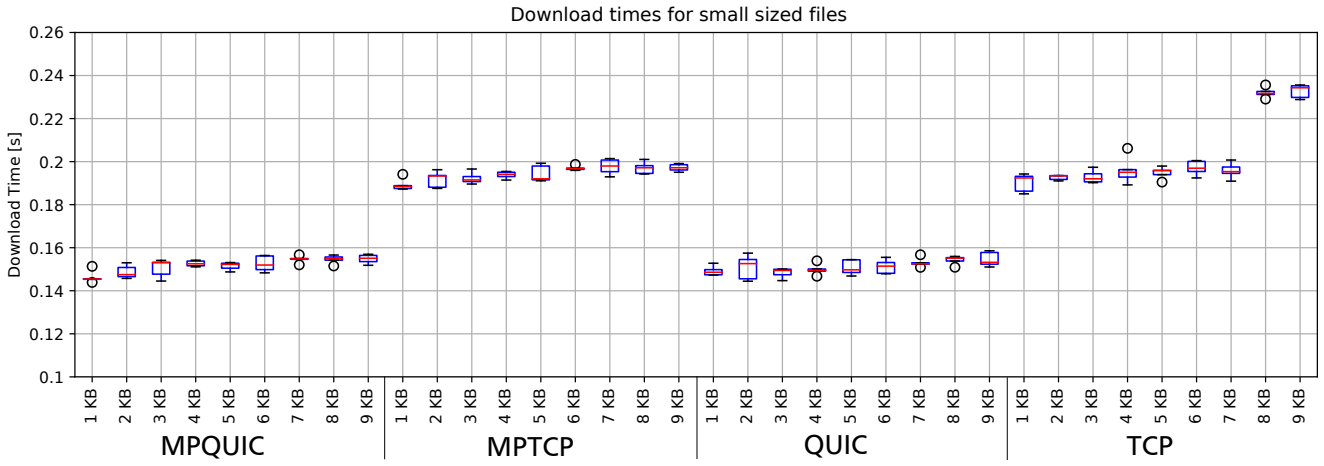


Figure 7.10.: Comparison of MPQUIC, MPTCP, QUIC and TCP for HTTP/2s requests of small files. The bandwidth of each subflow is 10 Mbps. The RTT is 44ms. TCP and MPTCP require additional time for the TLS handshake.

Next the download time for downloads of size between 10 KB and 90 KB is compared. Figure 7.11 reveals a unsteady behavior of all protocols. The reason is the congestion window size and its growth function, which only allows to send a particular amount of data each RTT. We notice that MPQUIC sends approximately twice as much data as QUIC in each round (i.g., Figure 7.11, red circles). This corresponds to the design of the decoupled congestion controller with individual congestion windows in MPQUIC. We further notice that the behavior of MPQUIC equals the behavior of MPTCP. When neglecting the different base values of the download time, MPQUIC and MPTCP show the same jumps (i.e., 20 KB - 30 KB, 70 KB- 80 KB) in the download time. We conclude that the growth of the congestion window of the multipath protocols is similar, at least for the tested files. This evaluation shows that MPQUIC's design leads to a similar behavior as that of the state-of-the-art MPTCP.

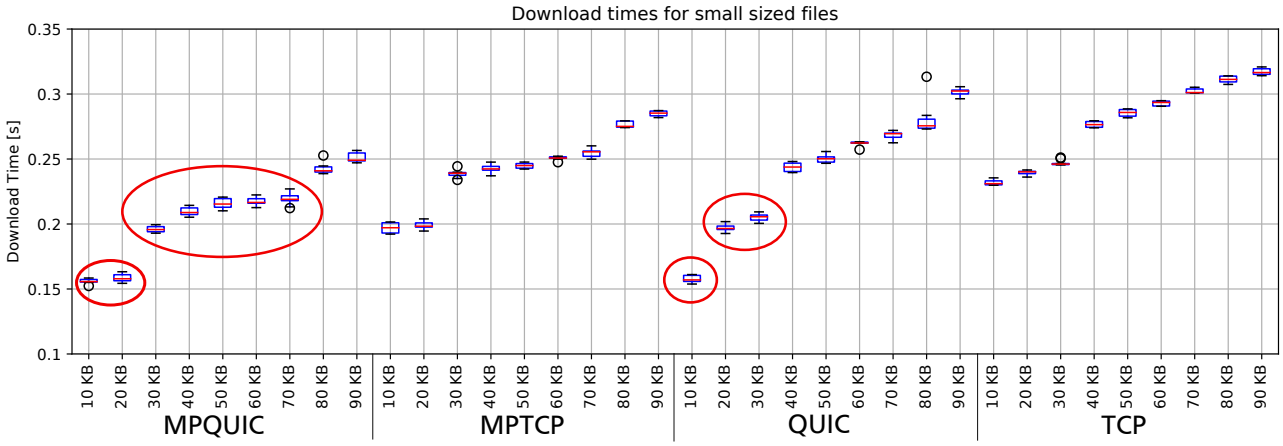


Figure 7.11.: Comparison of MPQUIC, MPTCP, QUIC and TCP for HTTP/2 requests of various files. The bandwidth of each subflow is 10 Mbps. The RTT is 44ms. The red circles (illustrated for MPQUIC and QUIC) indicate how much data can be sent in each *sending round*. The jumps between rounds (e.g., QUIC: 10 KB - 20 KB) originate from the congestion window size. Multipath protocols using two paths send more data in each round duo to the larger initial congestion window.

7.2.2 Heterogeneous Networks

Next the download time in heterogeneous networks is measured. We start with varying the bandwidth ratio of the additional path first and then vary the RTT ratio. For single path protocols, the bandwidth or RTT ratio of the additional path has no impact. Hence the figures only show QUIC and TCP with a bandwidth or RTT ratio of 1.0.

Bandwidth Heterogeneity

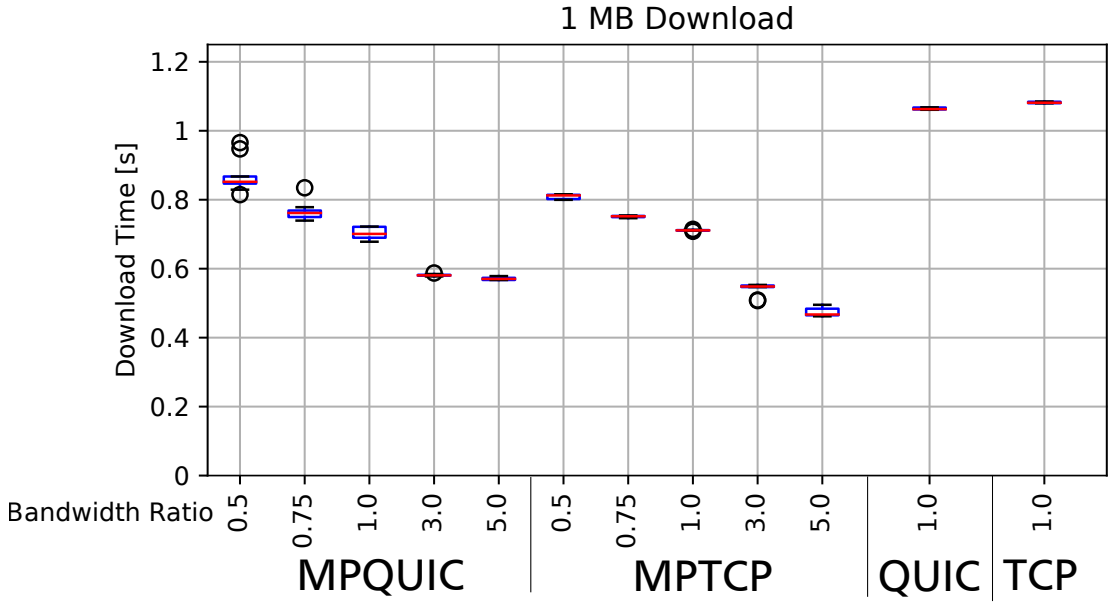


Figure 7.12.: Comparison of MPQUIC, QUIC, MPTCP and TCP for various bandwidth ratios. The bandwidth of the initial subflow is 10 Mbps. The RTT is 44ms for each path.

Again, we use a constant $RTT = 44$ms for both paths, a bandwidth of $B_1 = 10$ Mbps for the initial path and vary the bandwidth ratio B_2/B_1 of the first and second path. The comparison of MPQUIC, MPTCP,

QUIC and TCP for a 1 MB file download with different bandwidth ratios is shown in Figure 7.12. Both multipath solutions decrease the download time compared to their single path relatives. Further the decrease in download time with increased bandwidth ratio of MPQUIC is very similar to MPTCP. In a direct comparison of MPQUIC and MPTCP we notice that MPQUIC has higher download times than MPTCP. Unlike the results in Section 7.2.1, here MPQUIC performs slightly worse then MPTCP for bandwidth ratios different than 1.
A comparison for file requests of 100 KB and 100 MB is given in the appendices Section B.2.2 in Figure B.10 and Figure B.11 respectively. While for 100 KB, MPQUIC requires the lowest download time, compared to TCP, MPTCP and QUIC, for downloading 100 MB, MPQUIC and MPTCP are almost equal. Both outperform their single path relatives by far and show a similar speedup.

Round-Trip Time Heterogeneity

The 1 MB file download time for various RTT ratios $(2 \times D_{2b})/(R_1 - 4ms)$ by a constant $R_1 = 44ms$ and a bandwidth of 10 Mbps are shown in Figure 7.13. The behavior of MPQUIC and MPTCP as well as their behavior compared to their single path relatives are again very similar. The only notable difference is a higher variance in the download time for MPQUIC. We expect that the underlaying quic-go implementation, which is under active development and not performance optimized, causes the high variance. Note again, that we use the stable publicly available MPTCP Linux Kernel implementation, which is evaluated and tested. As the mean values of the MPQUIC and MPTCP download time are similar, we don't focus on this here. However in future work, the high variance must be examined.
Evaluations with files of size 100 KB and 100 MB are appended in Section B.2.2. Figure B.12 and Figure B.13 again strengthen our previous statements. A comparison of MPQUIC and MPTCP shows no notable difference when measuring the download time. While in case of a 100 KB download both multipath protocols show a slow down for a ratio of factor 3.0 and 5.0, a speedup of factor 2 is achieved when loading a 100 MB file. The performance for small as well as for large file is almost identical.

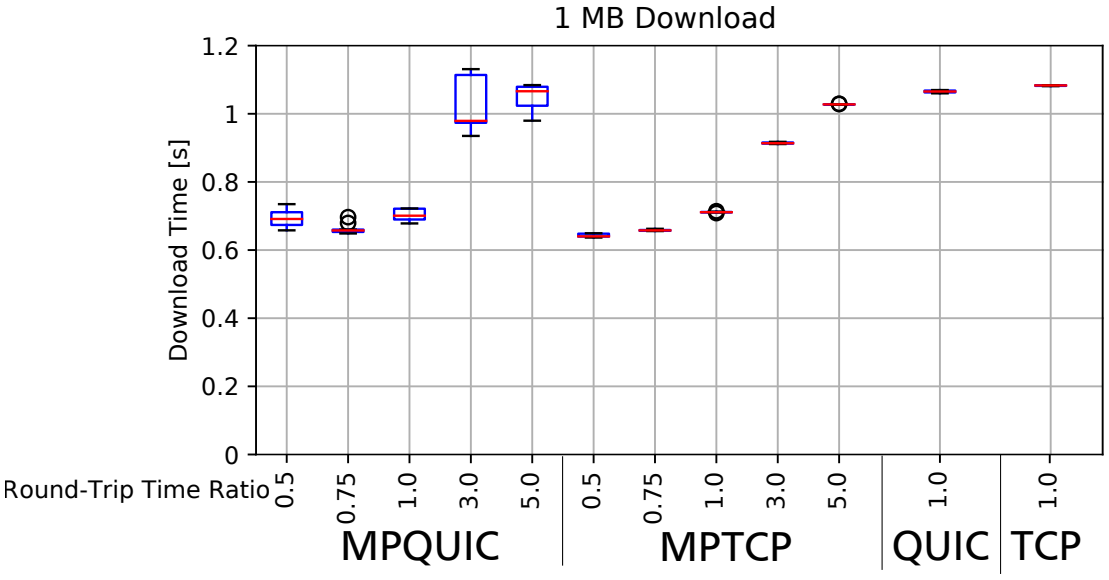


Figure 7.13.: Comparison of MPQUIC, QUIC, MPTCP and TCP for various RTT ratios. The RTT is 44ms for the initial path. The bandwidth of each subflow is 10 Mbps.

7.3 Loading Webpages

The previous experiments used to request a single file per run. However the most common use case for HTTP is loading of webpages. As QUIC originates in HTTP, the performance when loading webpages is

evaluated next.
A webpage usually consist of several files (e.g., style sheets, scripts, images) that need to be requested separately. Therefore webpages require several request after the main HTML is retrieved. The URLs for the required files are included in the main HTML. As described in Section 5.2, a simple parser is used to extract the URLs and the client sends HTTP requests for every extracted URL.
Based on the Alexa "*The top 500 sites on the web*" [1], we select two webpages for evaluation. When inspecting the most visited sites, we recognize that search engines (e.g., Google.com, Baidu.com.) and login-front pages (e.g., facebook.com) have a very high rating. However as we want to have several additional files (e.g., images) to be loaded, we select a Wikipedia article (total size of about 510 KB) for the evaluation. Wikipedia's frontpage is also in the Top 5 of the most visited sites. Additionally, we select a CNN news article (total size of about 8.2 MB) with several scripts and images. CNN is listed as the Top 2 of Alexa's most visited news sites. Both webpages, including additional files, are stored locally on the server host.
The page load time, beginning with the first message of the handshake, of the individual protocols for retrieving the selected webpages are measured and plotted in Figure 7.14. The evaluation confirms our previous measurements, showing that MPQUIC outperforms QUIC, TCP and MPTCP. As QUIC has a lower page load time than TCP, we conclude that the advance of MPQUIC over MPTCP is duo to its QUIC roots. Nevertheless the advantages of a multipath extension for QUIC are visible.

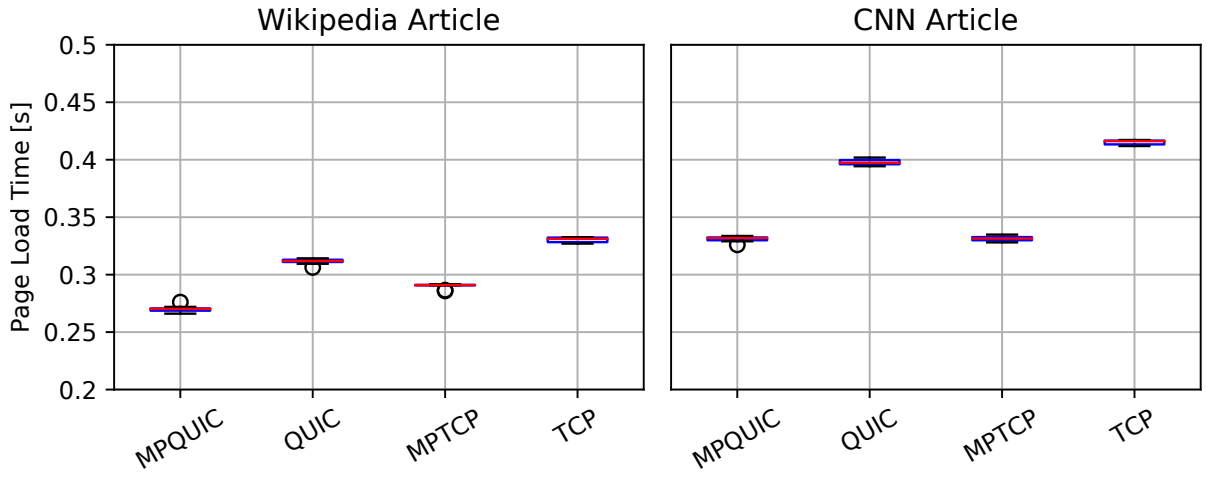


Figure 7.14.: HTTP/2 page load times with two homogeneous subflows. The bandwidth is 10 Mbps per subflow. The RTT of each subflow is 44ms.

7.4 Experimental Priority Scheduler

Beside the bulk load and webpage performance evaluations, we now provide a evaluation of the stream to subflow (m:n) scheduler advantages of MPQUIC. As discussed in Section 4.10 (also see Figure 4.14) MPQUIC enables the prioritization of individual application data in form of stream prioritization. MPTCP, in contrary, only *sees* a single stream of application data and processes these data in a FIFO order.
Accordingly we consider a setup where a 1 KB file of urgent data is requested, while loading a 10 MB low priority file at the same time. This illustrates a typical web page retrieval scenario, where a small JavaScript file is requested while images are being retrieved. The results of this setup in a heterogeneous network with a low delay (22ms) (e.g., WiFi) and a high delay (62ms) (e.g., cellular) path are shown in Figure 7.15 . The download time of MPQUIC and MPTCP for the 1 KB high priority file is on the left. The download time of the large low priority file is shown on the right. We observe that MPQUIC is able to transmit the high priority data significantly faster than MPTCP. MPTCP's single stream view results in

[1] https://www.alexa.com/topsites, retrieved 11/2017

a download time of 400ms for the high priority file. MPQUIC recognizes the application's priority and sends the 1 KB file as soon as possible, achieving a download time of 150ms on average.
Note that for this setup, we implemented an experimental stream scheduler, that prioritizes the stream of the 1 KB file. For subflow scheduling the default MinRTT scheduler is used. According to the m:n scheduler design, this corresponds to $m = 2$ (i.e., one stream for the small, one stream for the large file, when neglecting the header and crypto stream) and $n = 2$ (i.e., high delay and low delay subflow.) for MPQUIC. Furthermore we adjusted the HTTP/2 scheduler on top of MPTCP to prioritize the 1 KB file. This ensures, that the HTTP entity writes the 1 KB file on the MPTCP stream as soon as possible.

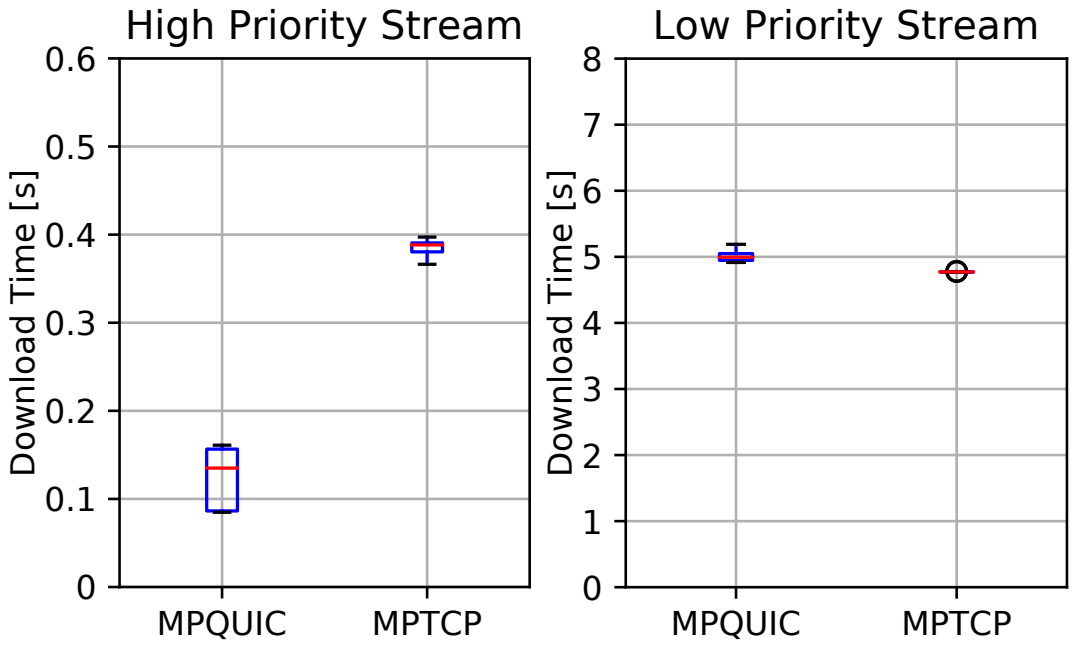


Figure 7.15.: In contrast to MPTCP, MPQUIC provides the ability to distinguish application specific stream priorities (here in the context of web page JavaScript objects vs. images).

7.5 Conclusion

Experiments in homogeneous as well as in heterogeneous networks show that MPQUIC's performance, with respect to the download time and speedup, is very similar to MPTCP. QUIC provides advantages over TCP when requesting small files(i.e., 100 KB and less), duo to the faster connection establishment. MPQUIC is able to transfer this advantages to multipath environments and shows a performance advantages over MPTCP when requesting small files. For larger files, MPTCP and MPQUIC provide the same speedup compared to their single path relatives. Overall, bulk data loading experiments reveal a similar behavior of MPQUIC and MPTCP. When now considering the faster connection establishment of QUIC and MPQUIC, we conclude that MPQUIC has a slight performance advantage over MPTCP.
The stream scheduler experiment (Section 7.4) highlight a major advantage of MPQUIC over MPTCP. The ability to consider multiple application data streams independently enables the involvement of application/user preferences into the scheduling decision.

8 Conclusion and Future work

We presented a multipath extension for the novel transport protocol QUIC. The broad design space for MPQUIC enables many different designs, that may be suitable for specific situations or use cases. However the focus of the provided solution is a multipath protocol extension for general purpose. For MPQUIC we designed a decoupled packet number space to avoid missing packet ambiguity. Based on the assumption of different bottlenecks when using multiple paths, each subflow is observed by its own congestion controller, managing per subflow specific stats. The detection of lost packets is also a subflow dependent issue, however MPQUIC allows retransmissions on an arbitrary subflow. The encryption and the flow control acts on connection level and is not aware of multiple subflows. For the individual components, the RTT-statistics have been adjusted, depending on their subflow awareness. Finally the Lowest-RTT-First scheduler decides, when to use the subflows.
Extending QUIC for multipath support was benefited greatly by QUIC's natural design. The presence of a `ConnectionID`, used as connection identified, in combination with the reserved `Multipath Flag` makes subflow establishment and the design of a new packet type, the `Announcement Packet`, very easy. Hereby subflows are used after only a half RTT by Announcement Receivers. Compared to MPTCP, which requires a full TCP handshake for subflow establishment, this is a significant improvement. Moreover MPQUIC's stream multiplexing enables stream to subflow (m:n) scheduling, whereas MPTCP only provides a single stream and thus only (1:n) subflow scheduling. First experiments with an experimental scheduler showed MPQUIC's ability of fine-grained scheduling by drastically reducing the download time of a urgent file. More general comparisons of MPQUIC and MPTCP revealed a similar behavior of the protocols. They both benefit from additional bandwidth and suffer from high delay subflows. The speedup of MPTCP over TCP is almost identical to the speedup of MPQUIC over QUIC. However, the faster connection establishment gives QUIC and MPQUIC a slightly reduces download time for small files, compared to TCP and MPTCP respectively.
Although MPQUIC already proved its ability to utilize additional bandwidth and thus achieve download times similar to the de facto multipath protocol MPTCP, there is still space for further optimization and considerations. First of all, the congestion control, which currently ignores fairness considerations, needs to be optimized for multipath usage. Congestion controllers are still developing and new solutions are proposed steadily. Furthermore we proposed an approach for subflow overreaching RTT-statistics, used by the flow controller and an RTT estimation approach for Announcement Receivers. Both approaches were not analyzed in a large-scale and thus need to be considered in future evaluations. Additionally as MPQUIC is designed for general purpose, wide deployments and evaluations for heterogeneous application scenarios, spanning from web to video retrieval, must be accomplished.
The most important aspect for future work is optimized utilization of the novel scheduling possibilities provided by MPQUIC. Retransmission may be pushed on particular subflows (e.g., lowest RTT) to reduce head-of-line blocking. Stream data of applications on top of MPQUIC, may be scheduled with respect to preferences and priorities. We envision a rich variety of scheduling flavors, that hugely benefit the user experience.

Bibliography

[1] Aditya Akella, Bruce Maggs, Srinivasan Seshan, Anees Shaikh, and Ramesh Sitaraman. A measurement-based analysis of multihoming. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 353–364. ACM, 2003.

[2] Christian Artigues. The resource-constrained project scheduling problem. *Resource-Constrained Project Scheduling: Models, Algorithms, Extensions and Applications*, pages 19–35, 2010.

[3] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540 (Proposed Standard), May 2015. ISSN 2070-1721. URL `https://www.rfc-editor.org/rfc/rfc7540.txt`.

[4] Mike Belshe and Roberto Peon. Spdy protocol. *IETF, draft-mbelshe-httpbis-spdy-00*, 2012.

[5] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), May 1996. ISSN 2070-1721. URL `https://www.rfc-editor.org/rfc/rfc1945.txt`.

[6] Giovanna Carofiglio, Massimo Gallo, and Luca Muscariello. Optimal multipath congestion control and request forwarding in information-centric networks: Protocol design and experimentation. *Computer Networks*, 110:104–117, 2016.

[7] Lily Chen. Recommendation for key derivation through extraction-then-expansion. *NIST Special Publication*, 800:56C, 2011.

[8] Iffat Sharmin Chowdhury, Jutheka Lahiry, and Syed Faisal Hasan. Performance analysis of datagram congestion control protocol (dccp). In *Computers and Information Technology, 2009. ICCIT'09. 12th International Conference on*, pages 454–459. IEEE, 2009.

[9] Gregory Detal, Benjamin Hesmans, Olivier Bonaventure, Yves Vanaubel, and Benoit Donnet. Revealing middlebox interference with tracebox. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 1–8. ACM, 2013.

[10] M. Duke, R. Braden, W. Eddy, and E. Blanton. A Roadmap for Transmission Control Protocol (TCP) Specification Documents. RFC 4614 (Informational), September 2006. ISSN 2070-1721. URL `https://www.rfc-editor.org/rfc/rfc4614.txt`. Obsoleted by RFC 7414, updated by RFC 6247.

[11] Morris J Dworkin. Sp 800-38d. recommendation for block cipher modes of operation: Galois/-counter mode (gcm) and gmac. *Special Publication (NIST SP)*, 2007.

[12] Rescorla Eric. The Transport Layer Security (TLS) Protocol Version 1.3, 2017. URL `https://tools.ietf.org/html/draft-ietf-tls-tls13-21`.

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068 (Proposed Standard), January 1997. ISSN 2070-1721. URL `https://www.rfc-editor.org/rfc/rfc2068.txt`. Obsoleted by RFC 2616.

[14] S. Floyd and E. Kohler. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control. RFC 4341 (Proposed Standard), March 2006. ISSN 2070-1721. URL `https://www.rfc-editor.org/rfc/rfc4341.txt`.

[15] S. Floyd, E. Kohler, and J. Padhye. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC). RFC 4342 (Proposed Standard), March 2006. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc4342.txt. Updated by RFCs 5348, 6323.

[16] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural Guidelines for Multipath TCP Development. RFC 6182 (Informational), March 2011. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc6182.txt.

[17] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824 (Experimental), January 2013. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc6824.txt.

[18] Bryan Ford. Structured streams: a new transport abstraction. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 361–372. ACM, 2007.

[19] Alexander Froemmgen, Amr Rizk, Tobias Erbshaeusser, Max Weller, Boris Koldehofe, Alejandro Buchmann, and Ralf Steinmetz. A Programming Model for Application-defined Multipath TCP Scheduling. In *ACM/IFIP/USNIX Middleware*, 2017.

[20] Mario Gerla and Leonard Kleinrock. Flow control: A comparative survey. *IEEE Transactions on Communications*, 28(4):553–574, 1980.

[21] Ilya Grigorik. Making the web faster with http 2.0. *Queue*, 11(10):40:40–40:53, October 2013. ISSN 1542-7730. doi: 10.1145/2542661.2555617. URL http://doi.acm.org/10.1145/2542661.2555617.

[22] Ilya Grigorik. *High Performance Browser Networking: What every web developer should know about networking and web performance*. " O'Reilly Media, Inc.", 2013.

[23] K-J Grinnem Grinnem, Torbjorn Andersson, and Anna Brunstrom. Performance benefits of avoiding head-of-line blocking in sctp. In *Autonomic and Autonomous Systems and International Conference on Networking and Services, 2005. ICAS-ICNS 2005. Joint International Conference on*, pages 44–44. IEEE, 2005.

[24] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008.

[25] Ryan Hamilton, Janardhan Iyengar, Ian Swett, and Alyssa Wilk. QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2. *IETF, draft-hamilton-early-deployment-quic-00*, 2017.

[26] L Todd Heberlein and Matt Bishop. Attack class: Address spoofing. In *Proceedings of the 19th National Information Systems Security Conference*, pages 371–377, 1996.

[27] Benjamin Hesmans, Fabien Duchene, Christoph Paasch, Gregory Detal, and Olivier Bonaventure. Are TCP extensions middlebox-proof? In *Hot Middleboxes*, pages 37–42. ACM, 2013.

[28] J Iyengar and M Thomson. Quic: A udp-based multiplexed and secure transport. *draft-ietf-quic-transport-05*, 2017.

[29] Janardhan Iyengar and Ian Swett. QUIC Congestion Control And Loss Recovery. *IETF, draft-iyengar-quic-loss-recovery-01*, 2016.

[30] Arash Molavi Kakhki, Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove. Taking a long look at quic: an approach for rigorous evaluation of rapidly evolving transport protocols. In *Proceedings of the 2017 Internet Measurement Conference*, pages 290–303. ACM, 2017.

[31] Morteza Kheirkhah, Ian Wakeman, and George Parisis. MMPTCP: A multipath transport protocol for data centers. In *INFOCOM*. IEEE, 2016.

[32] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), March 2006. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc4340.txt. Updated by RFCs 5595, 5596, 6335, 6773.

[33] H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869 (Informational), May 2010. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc5869.txt.

[34] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc2104.txt. Updated by RFC 6151.

[35] Adam Langley and Wan-Teh Chang. QUIC Crypto, 2016. URL http://goo.gl/0uVSxa.

[36] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *SIGCOMM*, pages 183–196. ACM, 2017.

[37] Feng Lu, Hao Du, Ankur Jain, Geoffrey M Voelker, Alex C Snoeren, and Andreas Terzis. Cqic: Revisiting cross-layer congestion control for cellular networks. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, pages 45–50. ACM, 2015.

[38] Daniel Lukaszewski and Geoffrey Xie. Multipath transport for virtual private networks. In *10th USENIX Workshop on Cyber Security Experimentation and Test (CSET 17)*, 2017.

[39] Bishop Michael. Header compression for http/quic. *IETF, draft-ietf-quic-http-04*, 2017.

[40] Bishop Michael. Hypertext transfer protocol (http) over quic. *draft-bishop-quic-http-and-qpack-01*, 2017.

[41] Frederic P Miller, Agnes F Vandome, and John McBrewster. *Advanced encryption standard*. Alpha Press, 2009.

[42] J.C. Mogul and S.E. Deering. Path MTU discovery. RFC 1191 (Draft Standard), November 1990. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc1191.txt.

[43] Christoph Paasch and Sebastien Barre. Multipath TCP in the Linux Kernel. available from http://www.multipath-tcp.org.

[44] Christoph Paasch, Simone Ferlin, Ozgu Alay, and Olivier Bonaventure. Experimental evaluation of multipath tcp schedulers. In *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop*, pages 27–32. ACM, 2014.

[45] R. Peon and H. Ruellan. HPACK: Header Compression for HTTP/2. RFC 7541 (Proposed Standard), May 2015. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc7541.txt.

[46] J. Postel. User Datagram Protocol. RFC 768 (Internet Standard), August 1980. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc768.txt.

[47] J. Postel. Transmission Control Protocol. RFC 793 (Internet Standard), September 1981. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc793.txt. Updated by RFCs 1122, 3168, 6093, 6528.

[48] C. Raiciu, M. Handley, and D. Wischik. Coupled Congestion Control for Multipath Transport Protocols. RFC 6356 (Experimental), October 2011. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc6356.txt.

[49] Phillip Rogaway. Authenticated-encryption with associated-data. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 98–107. ACM, 2002.

[50] Jonathan Rosenberg. Udp and tcp as the new waist of the internet hourglass. *Work in Progress*, 2008.

[51] Jan Rüth, Christian Bormann, and Oliver Hohlfeld. Large-scale scanning of tcp's initial window. In *Proceedings of IMC '17, London, United Kingdom*, 2017.

[52] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 4507 (Proposed Standard), May 2006. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc4507.txt. Obsoleted by RFC 5077.

[53] J. Salowey, A. Choudhury, and D. McGrew. AES Galois Counter Mode (GCM) Cipher Suites for TLS. RFC 5288 (Proposed Standard), August 2008. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc5288.txt.

[54] Michael Scharf and Sebastian Kiesel. Nxg03-5: Head-of-line blocking in tcp and sctp: Analysis and measurements. In *Global Telecommunications Conference, 2006. GLOBECOM'06. IEEE*, pages 1–5. IEEE, 2006.

[55] W. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC 2001 (Proposed Standard), January 1997. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc2001.txt. Obsoleted by RFC 2581.

[56] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960 (Proposed Standard), October 2000. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc2960.txt. Obsoleted by RFC 4960, updated by RFC 3309.

[57] Paul Syverson. A taxonomy of replay attacks [cryptographic protocols]. In *Computer Security Foundations Workshop VII, 1994. CSFW 7. Proceedings*, pages 187–191. IEEE, 1994.

[58] Andrew S Tanenbaum and David J Wetherall. Computer networks. 4th. *Upper Saddle River*, pages 530–541, 2003.

[59] Andrew S Tanenbaum and David J Wetherall. Computer networks. 4th. *Upper Saddle River*, pages 683–693, 2003.

[60] Andrew S Tanenbaum and David J Wetherall. Computer networks. 4th. *Upper Saddle River*, page 527, 2003.

[61] Andrew S Tanenbaum and David J Wetherall. Computer networks. 4th. *Upper Saddle River*, pages 552–582, 2003.

[62] Andrew S Tanenbaum and David J Wetherall. Computer networks. 4th. *Upper Saddle River*, pages 495–611, 2003.

[63] Andrew S Tanenbaum and David J Wetherall. Computer networks. 4th. *Upper Saddle River*, pages 541–552, 2003.

[64] Wesley W Terpstra, Christof Leng, Max Lehn, and Alejandro Buchmann. Channel-based unidirectional stream protocol (cusp). In *INFOCOM, 2010 Proceedings IEEE*, pages 1–5. IEEE, 2010.

[65] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *NSDI*, volume 11, pages 8–8, 2011.

Appendices

A quic-go components

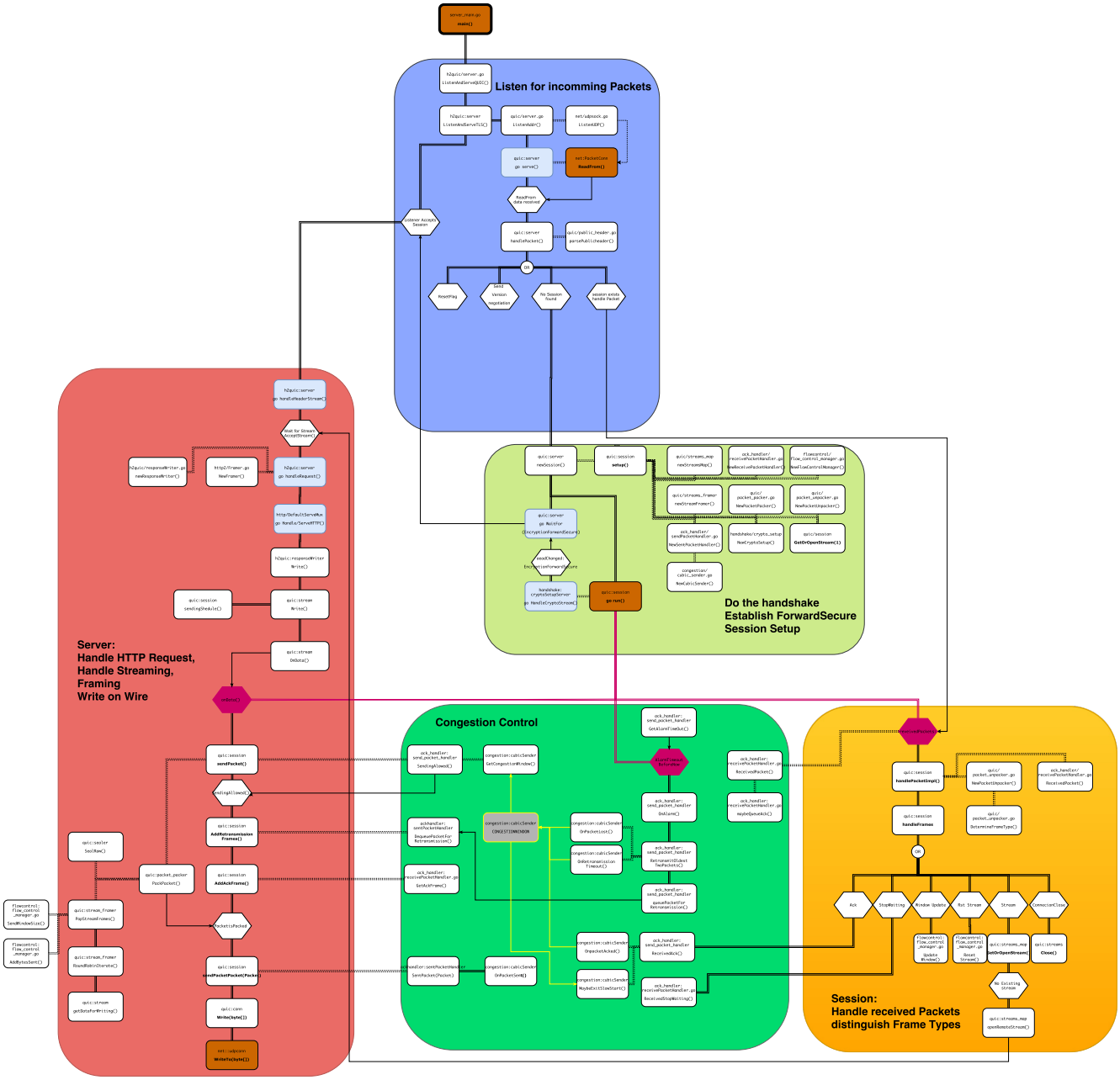


Figure A.1.: Components of quic-go

B Evaluation

B.1 Speedup in homogeneous networks

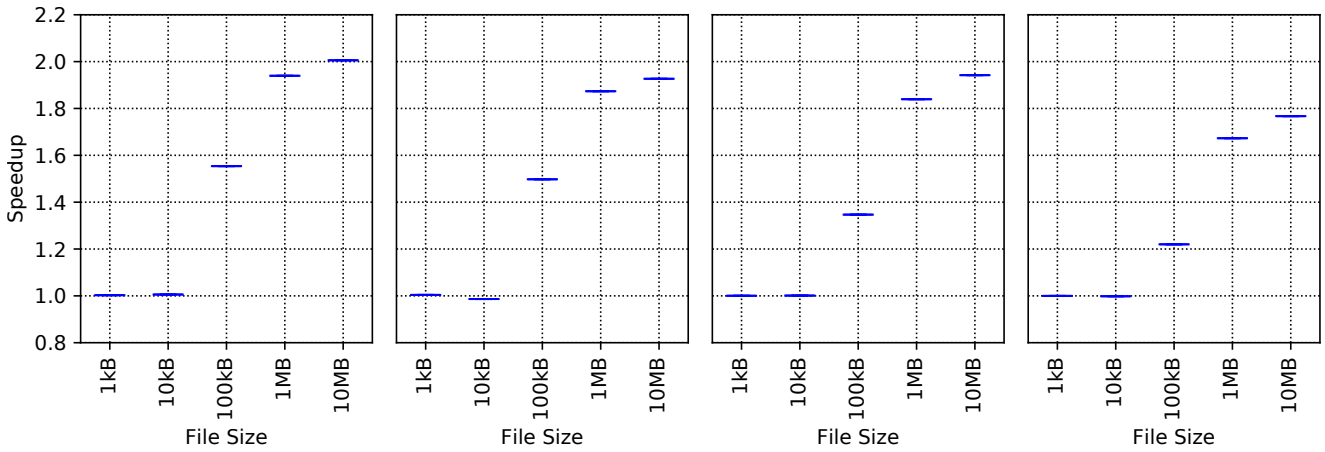


Figure B.1.: Speedup of MPQUIC using the MinRTT scheduler. Each path has a bandwidth of 1 Mbps. From left to right, the RTTs of both paths increase: 24ms, 44ms, 104ms, 204ms

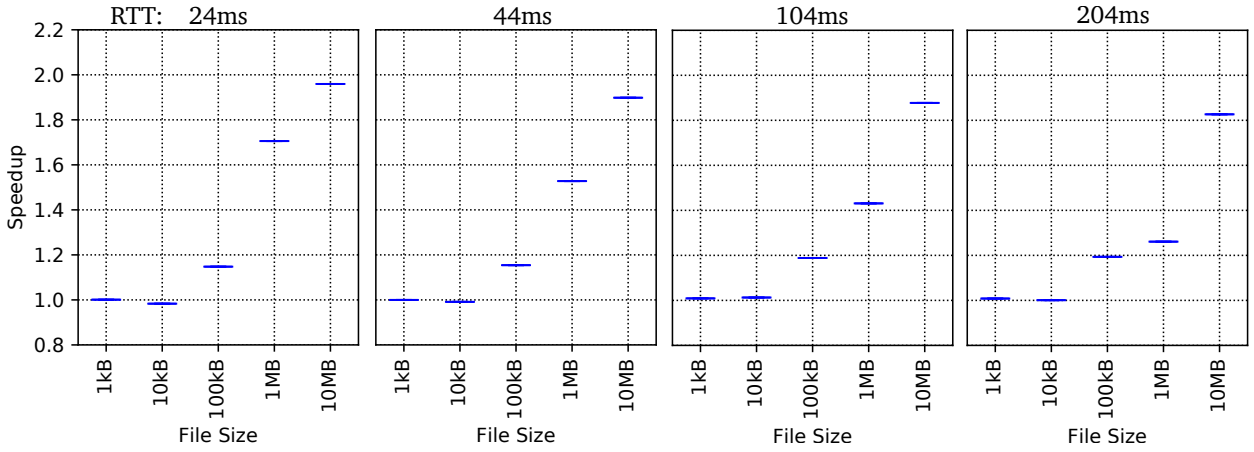


Figure B.2.: Speedup of MPQUIC using the MinRTT scheduler. Each path has a bandwidth of 10 Mbps. From left to right, the RTTs of both paths increase: 24ms, 44ms, 104ms, 204ms

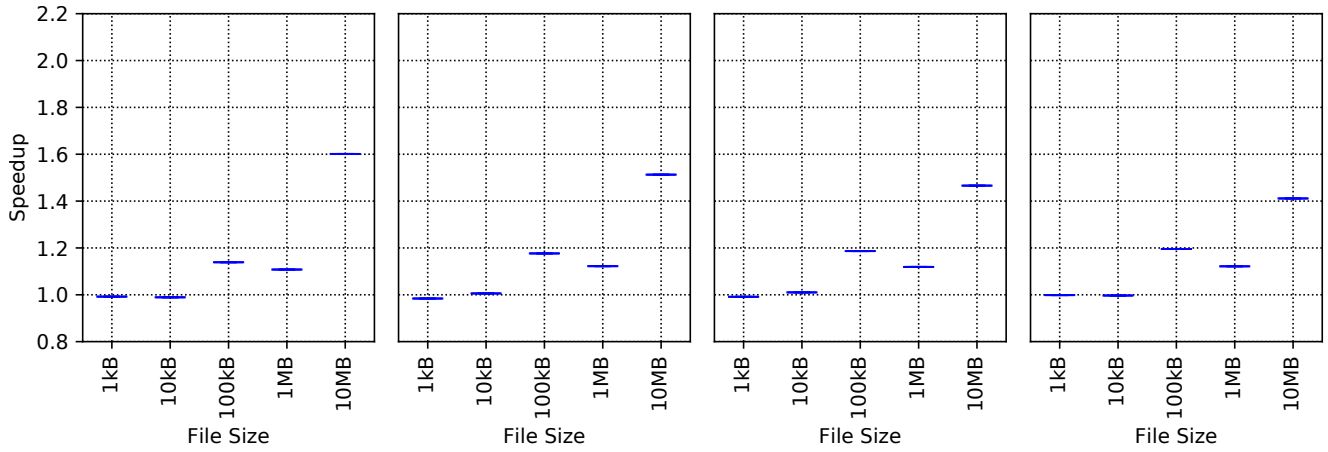


Figure B.3.: Speedup of MPQUIC using the MinRTT scheduler. Each path has a bandwidth of 100 Mbps. From left to right, the RTTs of both paths increase: 24ms, 44ms, 104ms, 204ms

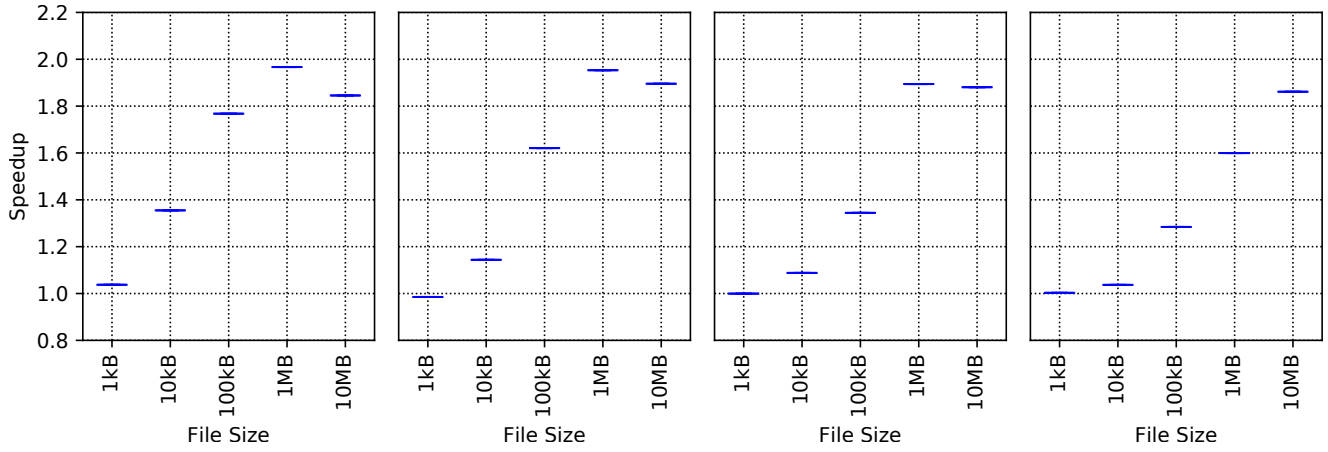


Figure B.4.: Speedup of MPQUIC using the RoundRobin scheduler. Each path has a bandwidth of 1 Mbps. From left to right, the RTTs of both paths increase: 24ms, 44ms, 104ms, 204ms

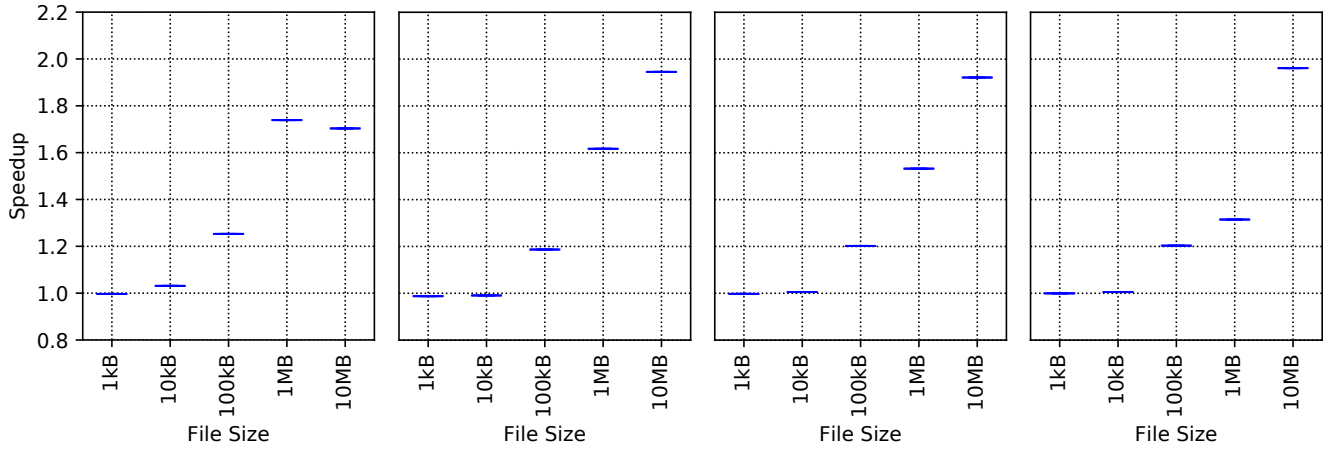


Figure B.5.: Speedup of MPQUIC using the RoundRobin scheduler. Each path has a bandwidth of 10 Mbps. From left to right, the RTTs of both paths increase: 24ms, 44ms, 104ms, 204ms

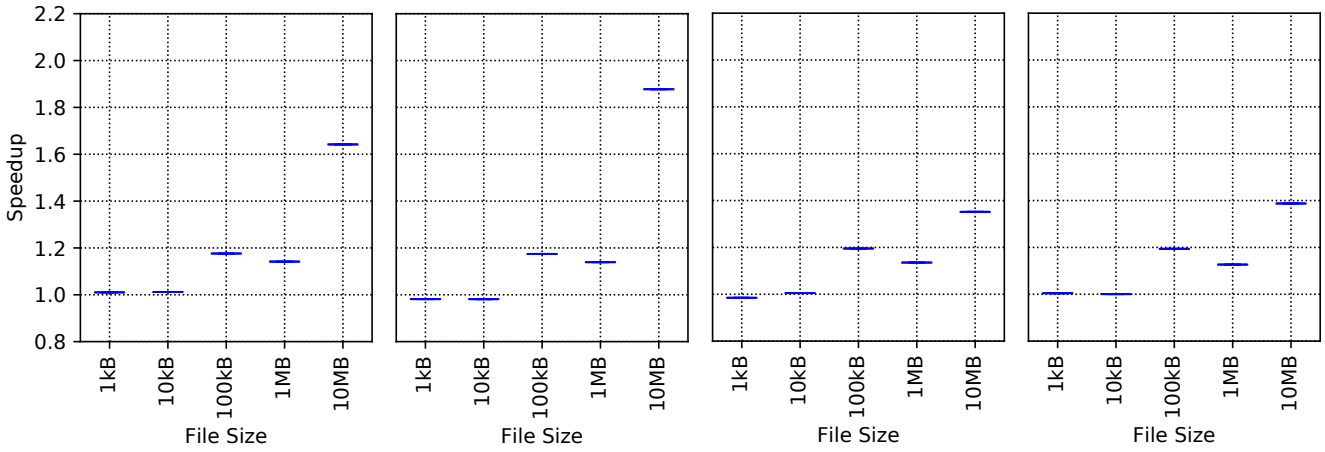


Figure B.6.: Speedup of MPQUIC using the RoundRobin scheduler. Each path has a bandwidth of 100 Mbps. From left to right, the RTTs of both paths increase: 24ms, 44ms, 104ms, 204ms

B.2 Comparison of QUIC, MPQUIC, TCP and MPTCP

B.2.1 Homogeneous Paths

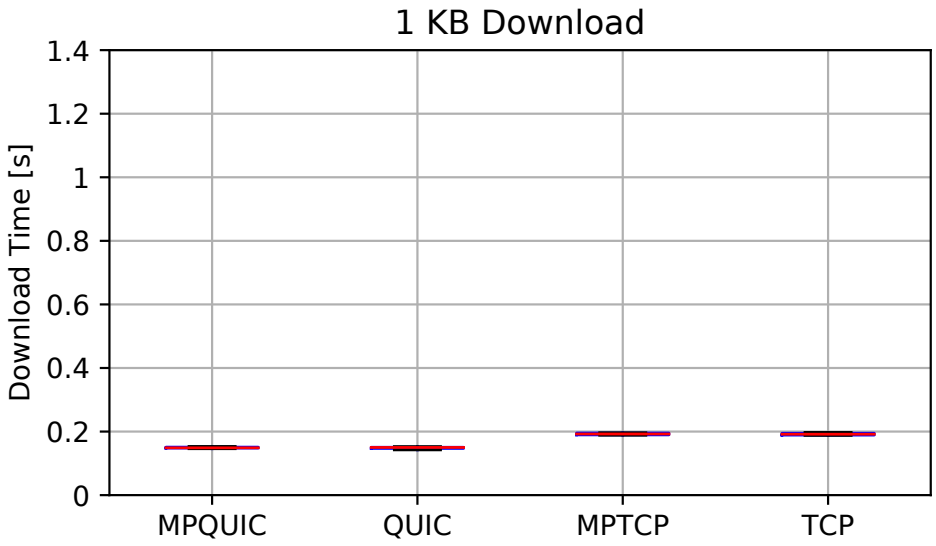


Figure B.7.: Comparison of MPQUIC, QUIC, MPTCP and TCP for HTTP/2 file downloads. The bandwidth of each subflow is 10 Mbps. The RTT is 44ms.

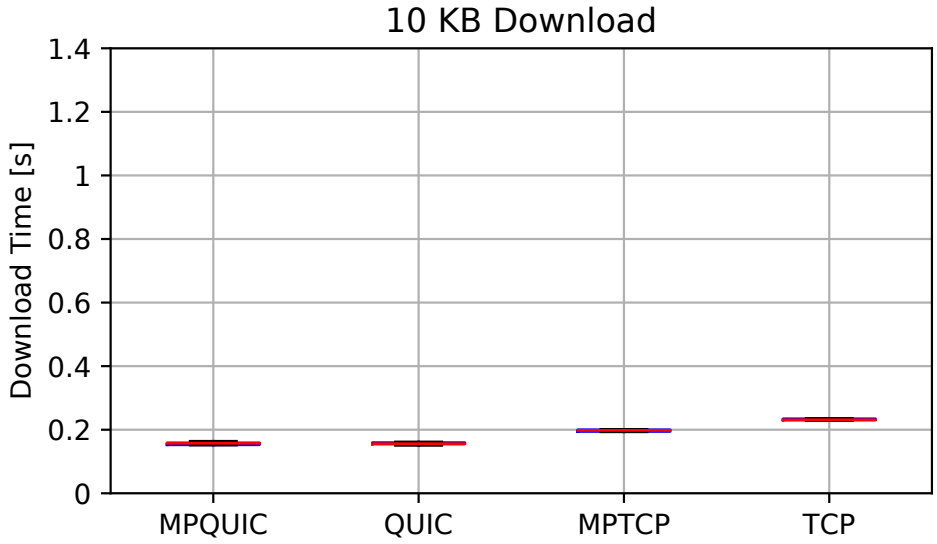


Figure B.8.: Comparison of MPQUIC, QUIC, MPTCP and TCP for HTTP/2 file downloads. The bandwidth of each subflow is 10 Mbps. The RTT is 44ms.

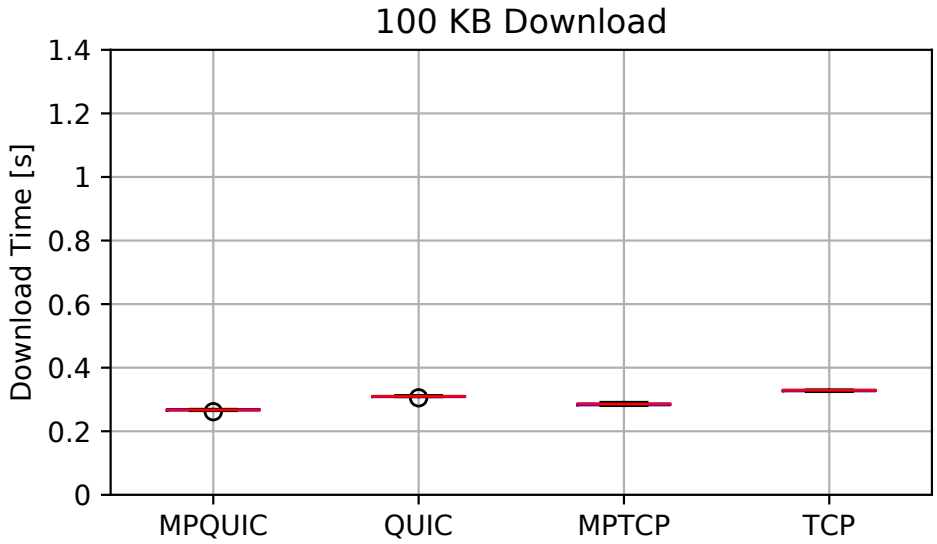


Figure B.9.: Comparison of MPQUIC, QUIC, MPTCP and TCP for HTTP/2 file downloads. The bandwidth of each subflow is 10 Mbps. The RTT is 44ms.

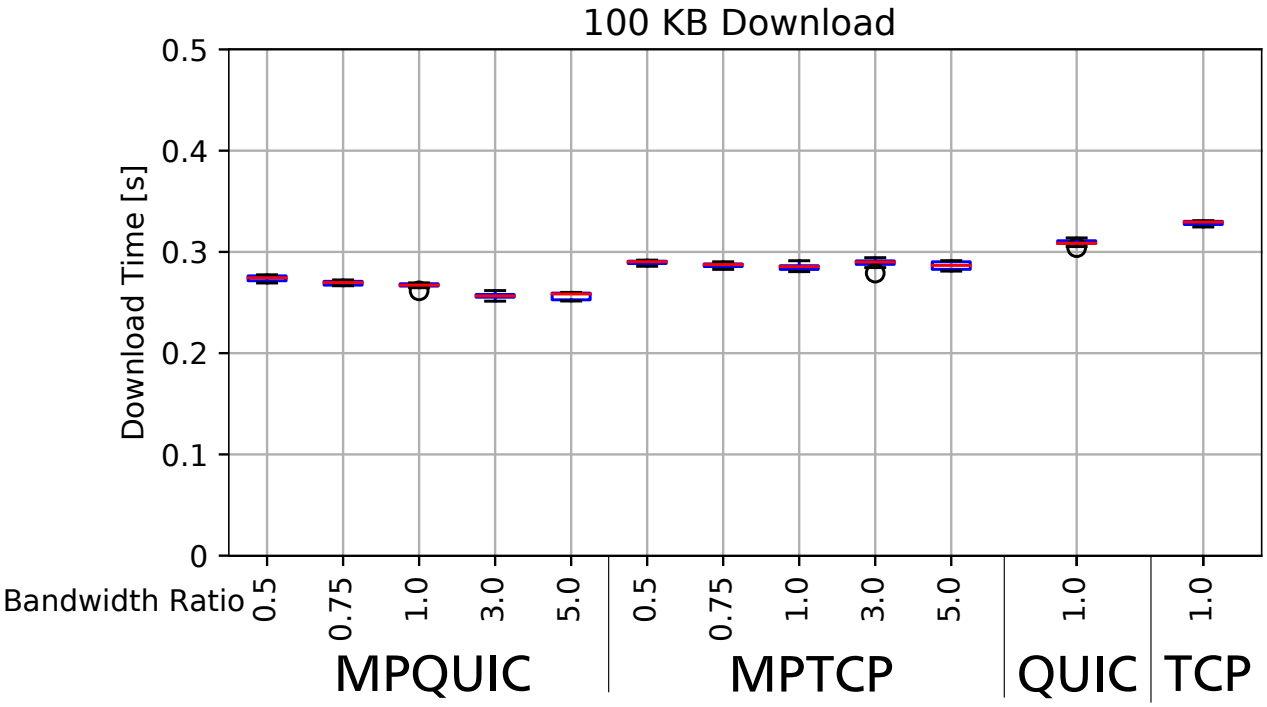


Figure B.10.: Comparison of MPQUIC, QUIC, MPTCP and TCP for various bandwidth ratios. The bandwidth of the initial subflow is 10 Mbps. The RTT is 44ms for each path.

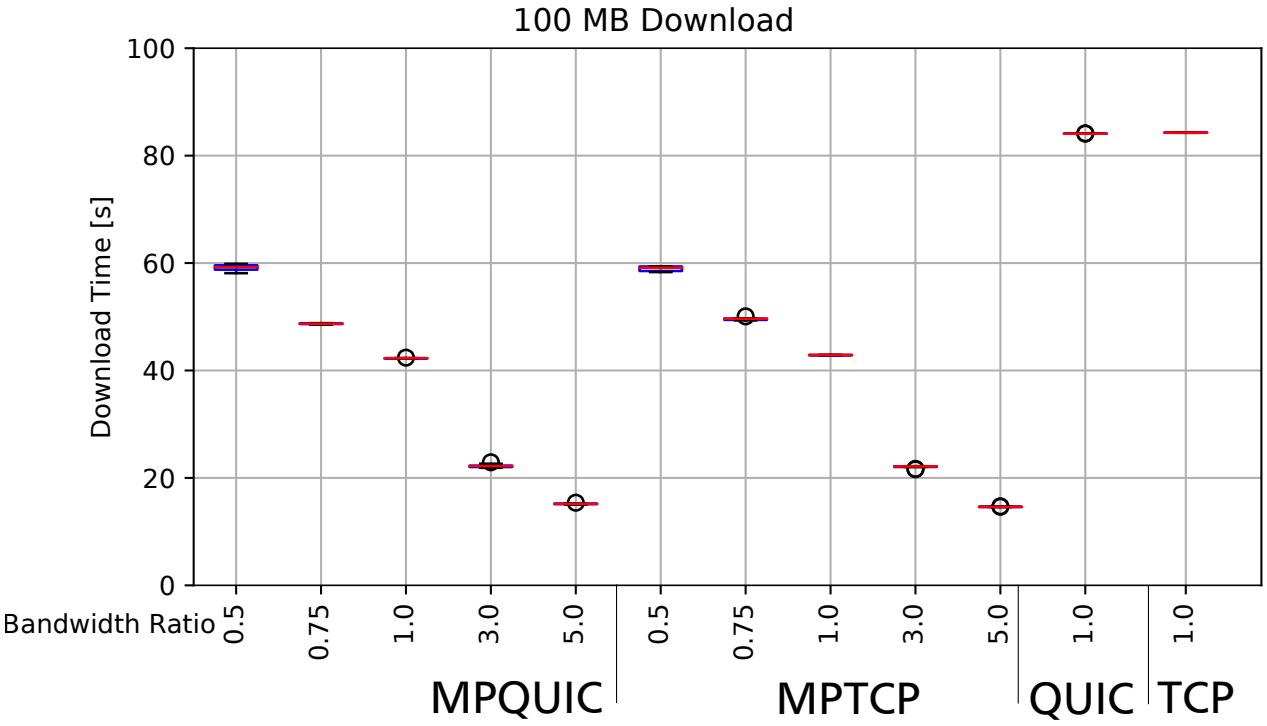


Figure B.11.: Comparison of MPQUIC, QUIC, MPTCP and TCP for various bandwidth ratios. The bandwidth of the initial subflow is 10 Mbps. The RTT is 44ms for each path.

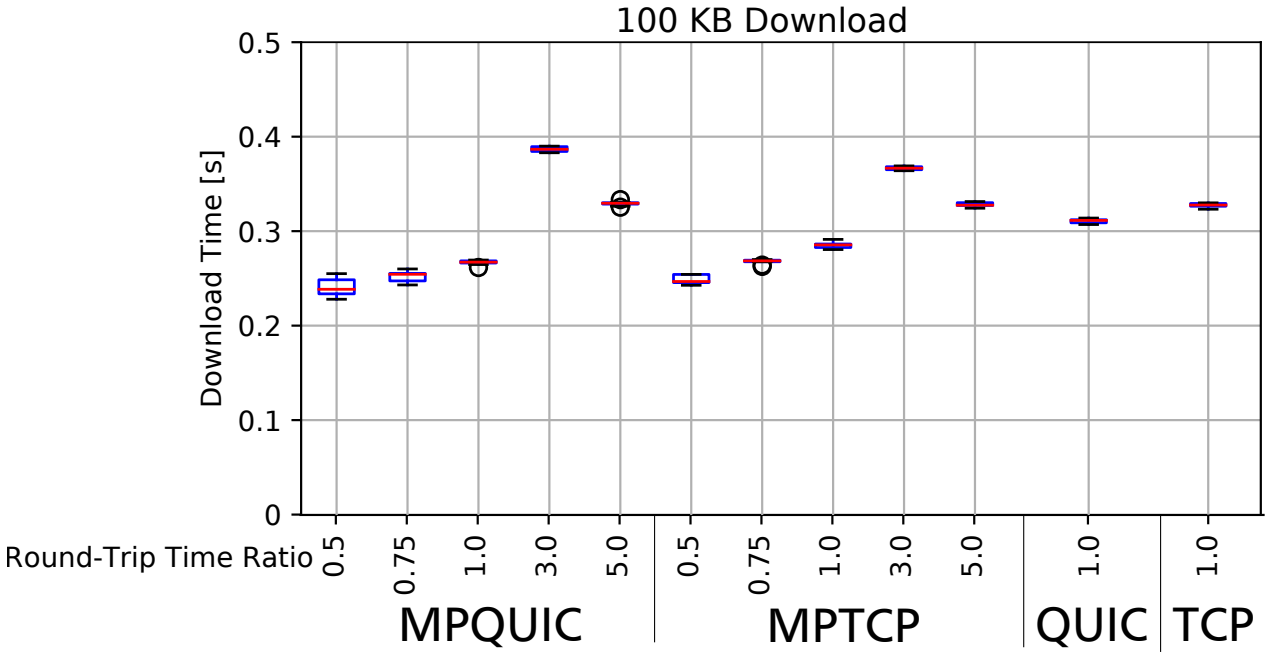


Figure B.12.: Comparison of MPQUIC, QUIC, MPTCP and TCP for various RTT ratios. The RTT is 44ms for the initial path. The bandwidth of each subflow is 10 Mbps.

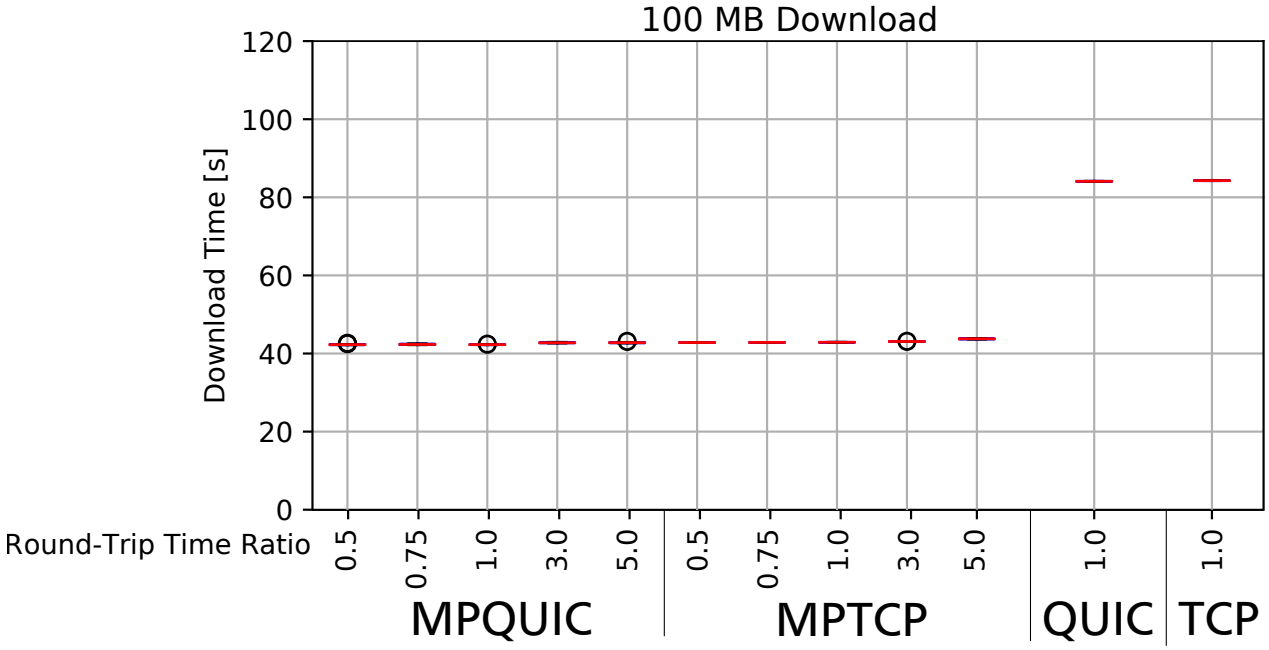


Figure B.13.: Comparison of MPQUIC, QUIC, MPTCP and TCP for various RTT ratios. The RTT is 44ms for the initial path. The bandwidth of each subflow is 10 Mbps.