Maastricht University

Faculteit Wetenschappen

School voor Informatietechnologie

master in de informatica

Masterthesis

MPAIOQUIC : On the design and implementation of MultiPath in QUIC

Tim Mesotten

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Peter QUAX

BEGELEIDER :

De heer Robin MARX

De transnationale Universiteit Limburg is een uniek samenwerkingsverband van twee universiteiten in twee landen: de Universiteit Hasselt en Maastricht University.

UHASSELT

KNOWLEDGE IN ACTION

www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

2020
2021

Maastricht University

Faculteit Wetenschappen

School voor Informatietechnologie

master in de informatica

Masterthesis

MPAIOQUIC : On the design and implementation of MultiPath in QUIC

Tim Mesotten

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Peter QUAX

BEGELEIDER :

De heer Robin MARX

[This page is intentionally left blank]

Abstract

A large part of connections on the Internet are TCP connections that allow pairs of endpoints to communicate and exchange data. However, the current design of TCP has a number of problems. First, by only allowing a single path to exist between two endpoints, the connection is limited to the throughput that is supported by the selected path. Additionally, TCP identifies the connection via a 4-tuple that is based on the IP-addresses that the endpoints used to establish the connection. If either endpoint loses an IP-address, the connection has to be terminated. Additionally, due to the design of TCP connections could suffer from HoL-blocking when packet loss is experienced.

A MultiPath extension was introduced to TCP to alleviate the first two problems. Here, a connection could use multiple paths simultaneously to allow communication between two endpoints. However, due to the inherent design of TCP not allowing for multiple streams to be used, a new protocol called QUIC has been developed. This protocol focuses on fixing many of the problems that TCP has, besides the mentioned ones. Due to the design of QUIC being different from TCP, a redesign on the concept of MultiPath is required. Several research teams and companies have already proposed different designs for the implementation which, combined with the fact that single-path QUIC mechanisms are reaching maturity, provides a series of discussion points on how MultiPath should be implemented in future versions of QUIC.

In this thesis, we provide an introduction to the concept of MultiPath in TCP and QUIC, and provide an implementation of the MultiPath concepts in AIOQUIC, based on the design of Quentin De Coninck and Olivier Bonavonture. Additionally, an evaluation of our implementation is given, and a discussion is provided on the current state of MultiPath design in QUIC.

Acknowledgements

This thesis was only possible because of the help and support from others. First, I would like to express my gratitude to my mentor, dr. Robin Marx, for his invaluable advice and feedback. Additionally, I want to thank Prof. dr. Peter Quax for giving me the opportunity to write this thesis. Lastly, I want to thank dr. Quentin De Coninck and Maxime Piraux from UCLouvain for their assistance in fixing the issues that were identified during the interop tests.

More personally, I would like to thank my mother for giving me the opportunity to attend university, and for supporting and pushing me throughout my whole life and education, even during very difficult times. Additionally, I would like to thank Brecht Bussers for proofreading this thesis. Finally, I would also like to thank my family and friends for the moments of humour and relaxation during the creation of this thesis.

Tim Mesotten, December 2020

Contents

Abstract **0**

Acknowledgements **0**

1 Introduction **2**
1.1 Purpose of this thesis . . . 3

2 Concept of MultiPath **4**
2.1 Multiple paths . . . 4
2.2 What is MultiPath? . . . 4
2.3 Requirements of MultiPath . . . 5

3 MultiPath TCP (MPTCP) **6**
3.1 MultiPath in TCP . . . 6
3.2 Requirements for MultiPath in TCP . . . 7
3.3 Initiating an MPTCP connection . . . 9
3.4 Using additional subflows . . . 10
3.5 Adding additional IP-addresses . . . 12
3.6 Removing an IP-address . . . 14
3.7 Transporting data . . . 15
3.7.1 Mechanism to send data over multiple paths . . . 16
3.7.2 An example of data transport . . . 18
3.7.3 Flow control and congestion control . . . 20
3.7.4 Middlebox interference . . . 21
3.8 Closing an MPTCP connection . . . 25
3.9 Problems of MPTCP . . . 27
3.10 MPTCP deployment . . . 28

4 Quick UDP Internet Connections (QUIC) **29**
4.1 Connection identifiers . . . 29
4.2 Connection migration . . . 32

5 MultiPath QUIC (MPQUIC) **36**
5.1 MultiPath in QUIC . . . 36
5.2 Requirements for MultiPath in QUIC . . . 37
5.3 Initiating a MPQUIC connection . . . 38
5.4 From using bidirectional paths to creating uniflows . . . 39
5.5 Using additional uniflows . . . 43
5.6 Adding additional IP-addresses . . . 49
5.7 Removing an IP-address . . . 54
5.8 Migrating uniflows . . . 59
5.9 Transporting data . . . 63
5.10 Closing a MPQUIC connection . . . 67
5.11 Problems of MPQUIC . . . 68
5.12 Differences between MPTCP and MPQUIC . . . 69

6 An MPQUIC implementation **71**
6.1 Aioquic . . . 71
6.1.1 Asyncio and SansIO . . . 71
6.1.2 Structure . . . 71
6.2 Implementing MultiPath . . . 74
6.2.1 Implementing multiple datagram endpoints . . . 74
6.2.2 Introducing additional IP-addresses to the connection . . . 75
6.2.3 Implementing the handling and writing of MP-specific frames . . . 76
6.2.4 Implementing uniflow classes . . . 77
6.2.5 Implementing the ability to send and receive data for multiple uniflows 78
6.3 Missing features . . . 79
6.4 Evaluation . . . 79

7 Evaluating MPAIOQUIC against other implementations **80**
7.1 MP Coverage . . . 80
7.1.1 PQUIC . . . 81
7.2 Performing Evaluations . . . 81
7.2.1 Handling other implementations . . . 81
7.2.2 PQUIC with MultiPath plugin . . . 83
7.3 Evaluation results . . . 88

8 Discussion and Future work **90**
8.1 Implementing MP Congestion Controller(s) . . . 90
8.1.1 Flow control . . . 90
8.1.2 Congestion control . . . 92
8.1.3 Achieving fairness . . . 95
8.1.4 Adding MultiPath to the mix . . . 96
8.2 Implementing packet schedulers . . . 103
8.2.1 Why is a packet scheduler required? . . . 103
8.2.2 MPTCP packet scheduling . . . 104
8.2.3 MPQUIC packet scheduling . . . 107
8.2.4 Acknowledgement strategies . . . 110
8.3 Usefulness of MPQUIC . . . 111
8.4 Future work . . . 112
8.4.1 Congestion control and packet scheduling in MPAIOQUIC . . . 113
8.4.2 Removing the receiving uniflows info section . . . 114

9 Conclusion **116**

List of Figures **119**

List of Tables **121**

Listings **122**

References **123**

1 Introduction

Nowadays, a large portion of Internet connections are TCP connections that allow two endpoints to communicate with each other and exchange data. One of the goals for the design of TCP was to allow the exchange of data to be performed in a reliable, in-order fashion. To accomplish this goal, the protocol allowed a single path to exist between endpoints over which the data is transported. Over the past decades since its deployment, TCP has seen many additions and changes being implemented. For example, the use of congestion controllers, and the proposals of extensions such as Explicit Congestion Notification and high performance extensions have been applied to further enhance the capabilities of TCP.

However, the design of TCP is not without inherent flaws. By only allowing a single path to exist between endpoints, that is also identified by the IP-address combination that both endpoints use to send data, a series of drawbacks were introduced. First, if an endpoint lost the IP-address for which it had established a connection with the remote peer, the connection has to be closed. Second, the throughput of the connection is limited to the transport capabilities of the selected path. Third, due to the design of TCP, connections could suffer from HoL-blocking when packet loss is experienced. Besides these three, there are additional problems that are not mentioned here.

To combat some of these drawbacks, an extension to TCP was introduced that proposed the concept of MultiPath, where multiple paths are used simultaneously in a connection to communicate with the remote peer. The extension presented the argument that modern systems often have multiple Network Interface Cards installed that allowed the system to connect with multiple networks. Based on this, systems would have multiple IP-addresses available that can be used to set up additional paths to communicate with a remote peer. The extension was designed with two use cases in mind, the first one being resilience to network failure, where a mechanism could allow the TCP connection to migrate to a different path and continue when an endpoint would lose an IP-address that was used to identify the connection. The second use case is the ability to use the multiple available paths simultaneously to increase the throughput of the connection. The design of the MultiPath extension also had overcome challenges that needed to be considered, because it had to be deployed in the current state of the Internet. However, while the extension aimed to solve some of the problems of TCP, no widespread use of MultiPath has been established yet. Besides this, the HoL-blocking problem of TCP proved to be very difficult to fix.

Thus, a new protocol called QUIC was designed on top of UDP. The protocol took a new perspective on providing a transport that allows data to be sent between endpoints in a reliable, in-order fashion. In the protocol's design a focus was placed on increased security and solving many of the existing problems of TCP. Due to the design of QUIC being so different from TCP, a complete redesign of the MultiPath concept has to be done to allow QUIC connections to use multiple paths simultaneously. For this reason, the introduction of the MultiPath design was postponed by the working group to a later version of QUIC. Nevertheless, there were some research teams and companies that had an interest in the use of MultiPath in QUIC. As a result, several MultiPath proposals have been given in the form of extensions.

1.1 Purpose of this thesis

Currently, the mechanisms of the single-path QUIC design are reaching maturity, and multiple discussions now take place on how the MultiPath concept should be implemented in the QUIC protocol. There already have been a number of studies that provided insights on the behaviour of different MultiPath QUIC implementations. From this, a series of different perspectives can be identified of which no perspective is found to be the best approach.

In this thesis, we present an introduction to the concept of MultiPath in both TCP and QUIC. Besides this, we also present our implementation of the MultiPath concept in the AIOQUIC library. The implementation is based on the mechanisms that are described in the MultiPath proposal of Quentin De Coninck and Olivier Bonaventure. Furthermore, we also evaluated our implementation and the proposed mechanisms in the form of interoperability tests with other QUIC implementations, such as the PQUIC implementation. Finally, we also provide a discussion about the current state of MultiPath design in QUIC.

This thesis is organised as follows, it first introduces in section 2 the concept of MultiPath in a general format. It then describes in section 3 the operations of MP in TCP, followed by the introduction of QUIC in section 4, and a description of MP operations in QUIC in section 5. In section 6 a description is given of our MP implementation in AIOQUIC, followed by the evaluation of our implementation in section 7. Finally, it discusses additional parts of the MP operations and the current state of MP in QUIC in section 8.

2 Concept of MultiPath

The concept of MultiPath is based on the fact that modern systems often have multiple IP-addresses available that can be used to connect to a remote peer on the Internet, resulting in multiple paths that can be used. This is possible because systems often have multiple Network Interface Controllers (NICs) installed, where each NIC allows the system to connect to a network and receive an IP-address. Thus, if a system has multiple NICs, it can connect to multiple different networks simultaneously, resulting in multiple IP-addresses being available to the system. For example, a modern laptop can connect to both a WiFi-network and a wired network simultaneously, resulting in two available IP-addresses, A1 and A2.

2.1 Multiple paths

If a system with multiple IP-addresses wants to connect to a remote peer, it can choose either address to do so. For example, if a system such as a laptop wants to connect to a remote webserver, it could select IP-address A1. The connection attempt would then create a path between the laptop and the webserver. This path is used by the connection to transport data between the endpoints, and it exists out of a sequence of physical links that propagate packets towards the next link that is one hop closer to the destination. However, if the laptop chose IP-address A2 to connect to the webserver, a completely different sequence of links could be used, resulting in a different path. When both systems have multiple IP-addresses available, multiple paths can be used. From the example, if the webserver also had two IP-addresses available, B1 and B2, a total of four unique paths could be created between the laptop and the webserver. See Figure 1 for a visualisation.

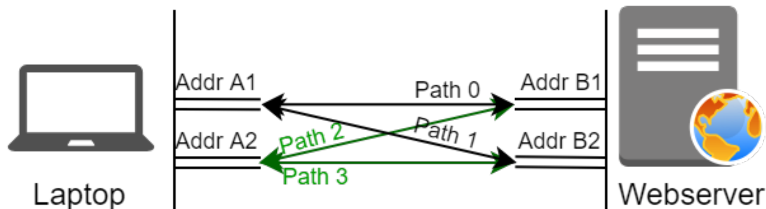


Figure 1: Potential paths for MultiPath.

2.2 What is MultiPath?

MultiPath aims to create a single connection between two endpoints where multiple paths can be used simultaneously if available, each carrying a flow of data between the endpoints. By doing this the connection can optimize the use of available resources in a network, as each of the paths have their own physical route between the peers and have their own capacity to transmit data. Combining these paths in a single connection allows for bandwidth aggregation, which in turn improves the total throughput of data. MultiPath also provides resilience to network failures within a connection because other paths can be used as a backup when problems occur on a certain path.

2.3 Requirements of MultiPath

Certain mechanisms need to be in place to enable the use of multiple paths in a single connection between two endpoints. The first mechanism would need to help an endpoint identify a connection for later use. This cannot be done via the address pair of the first path when a connection is set up, because a second path with a completely different combination of IP-addresses could also belong to the same connection. Thus, a different form of connection identification needs to be used upon creating the connection.

The second mechanism needs to be in place to allow an endpoint to signal their additional IP-addresses (and ports) to the peer. Both endpoints should only learn additional IP-addresses from their peer via the current existing connection, no additional external communication should be required. With this constraint, the signalling of an IP-address can be done in two ways during the connection. The first one being implicit by creating a new path that can be identified by the peer, while using an IP-address that the peer does not already know. The second one is more explicit by sending a message containing the IP-address' information. It is also required that the mechanism allows for the removal and signalling of IP-addresses that are no longer available. Finally, this mechanism also needs to be able to handle IP-address changes that are performed by NATs. One solution could be to use address IDs to indicate which IP-address is being handled.

The third mechanism would allow an endpoint to add a new path to an already existing connection. Some form of signalling needs to be in place to link this path to an already identifiable connection, which is made possible by the first mechanism. It is important that a security measure is in place to prevent external attackers from adding rogue paths to the connection. Both endpoints need to be able to authenticate their peer when adding additional paths. One solution could be to use randomly generated keys that are exchanged upon creating the connection. Whenever an additional path is identified, a message that is generated based on these keys could be signalled to indicate that the other endpoint is the same one as the peer that set up the original connection.

The fourth mechanism needs to support the endpoint in splitting the data over the multiple available paths in a connection. Additionally, this mechanism also needs to support the receiver by providing enough control information to combine the split data that is received from these paths back to its original form. It is also possible for some data that was sent over a specific path to be lost during a connection. Thus, the mechanism needs to be able to cope with this loss and track what part of the data has been sent over which path.

Finally, some additional requirements to security and reliability need to be adhered. For example, a mechanism needs to be in place that closes the connection when it detects problems that cannot be recovered from. Furthermore, some form of congestion control should be in place to prevent the paths from aggressively competing with other connections over the shared resources of a network. Additionally, it is a good practice that the mechanism of MultiPath reside in the transport layer, so that no other layers above and below it are affected, resulting in compatibility with older applications.

3 MultiPath TCP (MPTCP)

One of the goals for the design of TCP is to transport data in a reliable, in-order fashion between two endpoints [1]. To adhere to this goal the protocol allows a single path to exist between the endpoints over which data is transported during a connection. This single path (and thus, the TCP connection) is coupled to a 4-tuple by both endpoints, which is defined by the IP-address and port of both endpoints, and is used to identify a connection over which packets can be sent and received. The sender uses the 4-tuple to fill in the source and destination header values of a packet, while the receiver uses the 4-tuple to identify which connection a received packet belongs to. See Figure 2 for a visualisation.

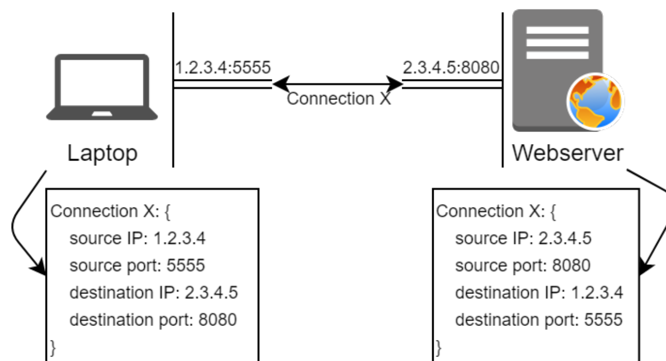


Figure 2: An example TCP connection.

This design of a TCP connection has led to additional extensions being proposed, such as Selective ACKs [2], Explicit Congestion Notification [3] and high performance extensions [4]. These extensions aim to improve the throughput of data by optimizing the use of available resources on a single path. However, further improvements can be made upon evolving the idea of using a single path towards one where multiple paths can be used.

In the current design of TCP only a single path is allowed to be used during a connection, and whenever a change in the 4-tuple is detected, the connection needs to be closed as neither endpoint can identify a connection for that new 4-tuple. An extension, called MultiPath TCP, has been proposed which enables a TCP connection to operate across multiple paths simultaneously, each carrying a flow of packets (a "subflow") [5]. The extension provides mechanisms to signal, set up, manage, and close additional subflows, as well as mechanisms to reassemble the data that is transported over these subflows, and a mechanism to communicate the availability of additional IP-addresses.

3.1 MultiPath in TCP

When regular TCP is used, a connection between two endpoints would create a single path by using one IP-address from each peer, resulting in all the other IP-addresses to remain unused. A MultiPath connection can detect these unused IP-addresses from a list of known local IP-addresses[1] and can select them to set up additional subflows with the peer. Each of these additional subflows can then be used to transfer data.

[1] This list is managed by listening for "address add/remove"-events

A given scenario: A modern laptop that can connect to a network over a wired connection, as well as a different local WiFi-network. The laptop would then have two IP-addresses available: A1 and A2, one for the wired connection and one for the WiFi-network. The laptop is going to download a large file from some remote webserver on the Internet. This webserver is also accessible via two IP-addresses: B1 and B2. A regular TCP connection could then exist on, for example, the address combination A1 and B1. A MultiPath TCP connection would see this as subflow 0. The extension could add additional subflows, for example, on the combination A2 and B1 (subflow 1). See Figure 3 for a visualisation.

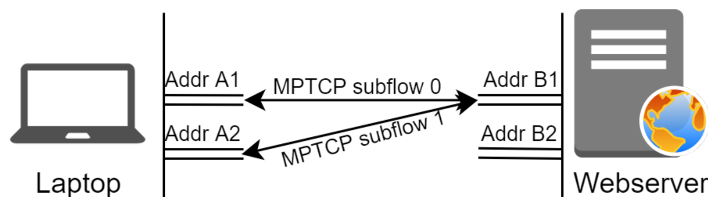


Figure 3: An example MPTCP connection.

3.2 Requirements for MultiPath in TCP

Before the concept of MultiPath can be implemented as an extension for TCP, some thought must be given to the current design of the TCP protocol. As previously stated, all MultiPath signalling must be done using an already existing connection between two endpoints. By comparing the changes and additions that other extensions have implemented, it is clear that MultiPath signalling can be done by using the provided options field in the TCP header of a packet. This header field has been introduced in the design of standard TCP to extend its reliability and performance by allowing the use of additional features that are not present in the standard implementation of TCP. The maximum size of the options field is also something that must be considered for MultiPath signalling, because it is possible for multiple extensions to be used on the connection, which also utilize the options field to communicate data. Thus, all MultiPath signalling should be as small in size as possible.

If MultiPath is to be implemented as an extension, it is entirely possible for a remote endpoint to not support the use of MultiPath at all. This can happen for various reasons, for example, a system could not have been updated for some time, resulting in the update that introduced the extension to not have been applied. Thus, whenever a new connection is initialized, an endpoint should confirm that its peer supports the MultiPath extension. Upon sending a SYN packet, the TCP options field can be used to signal that the initiating endpoint supports the MultiPath extension, and that it also wants to use the extension for the current connection.

If the other endpoint receives the connection attempt and does not support MultiPath it can safely ignore that option. If the other endpoint does support MultiPath and wants to use it, an equal signal can be added to the options-field of the responding SYN/ACK packet. If the initiating endpoint receives a packet with a TCP option that contains the expected MultiPath signal it can assume that the connection will be using additional MultiPath signals.

However, the MultiPath extension must also be implemented in such a way that it can be deployed in the current state of the Internet, which introduces additional problems for the extension to solve. Almost all these problems are introduced by the many middlebox implementations that run on the intermediate links of a connection. These middlebox implementations can differ between vendors, have their own bugs, and do not always have support for the proposed extensions of TCP. To avoid connection problems that could occur due to these middleboxes failing, the extension must be compatible with these middlebox implementations. A good solution for this problem is for the subflows of the extension to resemble a normal TCP connection as much as possible.

This problem is already solved for the most part because the MultiPath extension will create subflows that resemble a normal TCP connection, but can "secretly" transport MultiPath signals with the use of the options field. A single "kind"-value can be used to indicate that the option contains MultiPath signalling. To differentiate between the multiple signals, such as initiating a MultiPath capable connection, adding and removing additional IP-addresses, creating additional paths, and providing control information, a subtype can be used where each signal can have its own set of information that needs to be transmitted. That information can then be used by the corresponding mechanisms that are required for MultiPath. By using a single "kind"-option the problem is solved where middleboxes can perform different actions when confronted with option values that are unknown to them. Some could drop the option when forwarding the packet, while others would just ignore it. A single "kind"-option ensures that a non-supporting middlebox drops all MultiPath signalling entirely.

The other part of the problem is that middleboxes can perform actions such as segmenting and coalescing packets, which results in the modification of header values. The TCP options could be changed, stripped, or copied across multiple packets. The MultiPath extension relies on data to be transmitted in a specific way, based on the provided control information. If this information is modified or removed it can lead to problems when reassembling the data on a receiver's end. Whenever an endpoint detects problems that could have been caused by a middlebox it can perform certain actions to prevent the issues from breaking a connection. The endpoint can close the subflows where these issues occur, it can even perform a fallback to prevent connection loss.

A fallback mechanism that is provided by the extension can be useful for situations where a problem occurs that cannot be recovered from. For example, a situation can occur where a middlebox allows all options to be propagated when a new TCP connection is set up using the three-way handshake. In this case, the fist subflow of the connection is created while no other subflows exist yet. But once the connection has been established, that one middlebox now only propagates known TCP options, and drops the MultiPath signalling entirely. Both endpoints now have a connection where they assume that MultiPath signalling is supported, but no signalling is received from the peer. Upon detecting this, a fallback operation can be performed where the endpoints no longer perform MultiPath operations at all and continue to handle the connection as a regular TCP connection, thus only using a single path to transport data.

The following subsections explain how each of the mechanisms required for MultiPath operations are applied in the current design of the TCP protocol.

3.3 Initiating an MPTCP connection

A MultiPath TCP connection starts in the same way as a regular TCP connection. It is initiated via the three-way handshake that uses the SYN-, SYN/ACK- and ACK-packets. Each of these packets contain an additional TCP option called MP_CAPABLE, which tells an endpoint that the peer supports the MultiPath extension. With this option keys are exchanged during the handshake, along with additional flags that indicate whether checksums are required, and which cryptographic algorithm is negotiated for the authentication of additional subflows. The keys are chosen by each endpoint and are later used by the receiver of the key to authenticate additional subflows that need to be added to the established connection. If no such keys are exchanged, it would be impossible for the receiver to discern an additional subflow from a regular TCP connection, this is because those keys are used as an identifier for that specific connection.

From the scenario's perspective: The laptop connects to the remote webserver by sending a SYN-packet to the webserver carrying a MP_CAPABLE option, containing a key chosen by the laptop for this connection, and flags to indicate that it wants to use checksums and the default cryptographic algorithm, being HMAC-SHA1. The webserver replies with a SYN/ACK-packet that also carries a MP_CAPABLE option, containing a key chosen by the webserver, and flags to indicate that it also wants to use checksums and the default cryptographic algorithm.

The laptop stores that key and compares the flags. If at least one cryptographic algorithm is shared among both endpoints it will send back an ACK-packet containing both the communicated keys, along with the negotiated flags that were indicated in the previous two packets. In this case these flags indicate that both checksums and the default cryptographic algorithm will be used for the remainder of the connection, across all subflows. From now on, subflow 0 is established between the laptop and the server. See Figure 4 for a visualisation.

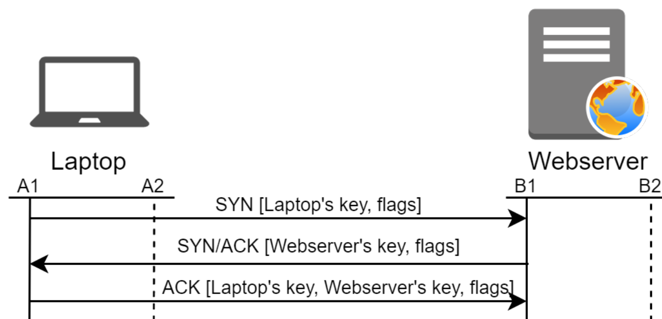


Figure 4: Creating a MPTCP connection.

However, it is possible that the remote webserver does not support the MultiPath extension, or that one of the middleboxes on the path has stripped the MP_CAPABLE option. In this case the original sender will receive a plain SYN/ACK-packet with no MultiPath signalling in the options field. If this were the case in the scenario, the laptop would decide that the connection cannot support MPTCP and it would perform a fallback to a regular TCP connection. This would be done by sending a normal TCP ACK-packet to the remote webserver.

3.4 Using additional subflows

When an initial, MultiPath-supporting connection between two endpoints has been established, it is possible for either endpoint to set up additional subflows with their peer. These subflows can be set up on the condition that there are combinations of address pairs between the peers that have not been used by other subflows. Reusing a combination of IP-addresses would mean that both the new subflow and the old subflow, that is bound to that same address pair, would share the same path, thus resulting in also sharing the path's capabilities to transmit data, which only results in both subflows competing for that path's resources.

To link a new subflow to an existing connection, some form of identification of a connection needs to be in place. The extension already provides an exchange of keys upon initiating the connection, thus the peer's key can now be used to signal that a newly created subflow is part of an already existing connection. For security, a peer's key will not be sent again by the initiator upon creating a new subflow. Instead, a hashed value of that key, a token, is transmitted. The receiver can generate a hash from its own key and check if the received token has the same value. If this is the case, the connection can be identified successfully and the subflow can be added to the connection by the receiver.

However, by only using a token, there is no way for the receiver of a subflow initiation to authenticate the initiator as being the peer with whom the receiver exchanged keys upon creating the connection. A malicious third party could have observed the creation of the new subflow and copy the token. The third party could also initiate a new subflow from its own IP-address and successfully trick the receiver into linking a rogue subflow to the connection that belongs to the key from which the copied token was generated. Thus, some form of authentication needs to be in place to prevent third parties from successfully linking rogue subflows to a connection.

In MPTCP this is done with the use of HMAC-messages that are generated with the cryptographic algorithm that has been negotiated upon creating the connection. By exchanging a HMAC-message from the receiver to the sender and vice-versa both endpoints can authenticate their peer as being the peer with whom they exchanged keys upon creating the connection. However, without additional protection these HMAC-messages can be used multiple times because the value will always remain the same when only using the key values as input. A malicious third party could observe the HMAC-message of a sender and copy it. Later the third party could initiate a new subflow from its own IP-address and authenticate themselves successfully on the receiver by using the copied HMAC-message. This allows the third part to perform a replay attack.

To prevent this, both endpoints should exchange random nonce values which should both be used to generate unique HMAC-messages on both endpoints. Additionally, both endpoints need to ensure that unique HMAC-messages generated from nonce values are only accepted once. If this additional measure is in place and the malicious third party tries to create an additional subflow with a copied HMAC-message, the receiver of that subflow initiation can identify that this HMAC-message has already been used, and it can reject the subflow initiation attempt.

Combining all this together, from the scenario's perspective: The laptop identifies that it has an additional IP-address available, A2. It also identifies that the address combination of A2 and B1 does not exist yet, so it decides to set up an additional subflow (subflow 1). The laptop sends a SYN-packet to the webserver carrying a MP_JOIN option, containing a token derived from the webserver's key, a random nonce that the laptop generated, an address ID that the laptop chose to identify its own IP-address A2 and a flag that indicates whether the laptop wants to use this subflow as a backup. The usage of an address ID is explained in section 3.5.

The webserver receives the SYN-packet and checks if the token matches any connection identifying key. If this is the case the webserver sends back a SYN/ACK-packet carrying a MP_JOIN option, containing a HMAC-message that is generated based on the exchanged keys, the laptop's nonce and a nonce that the webserver generated. The option also contains the webserver's generated nonce, an address ID that the webserver chose to identify its own IP-address B1 and a flag that indicates whether the webserver wants to use this subflow as a backup. At this point, the webserver knows that a subflow is being initiated, but the webserver does not know if this subflow is connected to the right peer.

The laptop receives the SYN/ACK-packet and checks if the server's HMAC-message is valid. If this is the case, the laptop will send back an ACK-packet carrying a MP_JOIN option, containing a HMAC-message that is generated based on the exchanged keys, the webserver's nonce and a nonce that the laptop sent previously. At this point, the laptop knows that the subflow initialization has been accepted, and that the peer, to whom it sent the SYN-packet, actually is the webserver. It can now safely add the subflow to the connection, but no data can be sent over the subflow for now.

The webserver receives this ACK-packet and checks if the laptop's HMAC is valid. If this is the case the webserver completes the handshake by sending a final plain ACK-packet back to the laptop. At this point, the webserver now knows that the subflow initialization attempt effectively comes from the laptop, so the webserver can now safely add the subflow to the connection and can send data on it after sending the ACK-packet. When the laptop receives the final plain ACK-packet it knows that the subflow initialization attempt with the webserver has been successfully completed and can start sending data on it. See Figure 5 for a visualisation and see Figure 3 for the resulting connection.

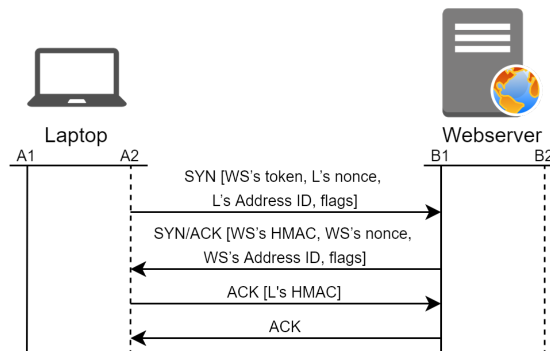


Figure 5: An additional subflow handshake.

It is important that the receiver of the subflow initiation attempt finalizes the handshake with a final regular ACK. This indicates that both endpoints have successfully authenticated each other, have created an additional subflow, and that this additional subflow has also been added to the previously established connection.

From an external perspective, the additionally initiated subflow looks like another regular TCP connection between the two endpoints. The sole difference being that a middlebox can see is that the packets contain extra TCP option values. Meaning that, even though the initial subflow is confirmed to be MultiPath supporting, the additional subflow is not guaranteed to support it as well. It is possible that a middlebox on the path of the additional subflow does not support the extension and could strip the MP_JOIN option from the handshake packets. If this would happen in the scenario, the laptop would receive a SYN/ACK-packet without the MP_JOIN option. In response the laptop would immediately close the subflow by sending a TCP RST Packet.

3.5 Adding additional IP-addresses

Additional subflows can be added to the connection on the condition that there exists an unused pair of IP-addresses between the two endpoints for which the connection exists. To identify unused pairs of IP-addresses, both hosts need to know their peers other IP-addresses. Therefore, hosts need to be able to communicate their additional IP-addresses to the peer. As defined by the requirements of MultiPath, the exchange of IP-addresses can be done in two ways, which are often combined in MPTCP. The first one being implicit by setting up an additional subflow, where the sender identifies the new IP-address with an address ID in the MP_JOIN option (see Figure 5).

The second way is more explicit, by using the ADD_ADDRESS option when sending a packet, which contains an IP-address and an address ID. Combining this second method with the first one can be effective when the peer resides behind a NAT. In this case, setting up a new connection will fail by timing out, but the ADD_ADDRESS option is still received by the peer on an already established subflow. Receiving the ADD_ADDRESS option works as an incentive for the receiver to open an additional subflow. In both ways the extension uses address IDs to identify IP-addresses. This is done to communicate about an IP-address without having to check if the IP-address has been altered by middleboxes. There is a special case that can occur when both endpoints reside behind a NAT, where setting up a new connection will also fail by timing out. This will happen in both directions, thus no subflow can be set up using that particular combination of IP-addresses.

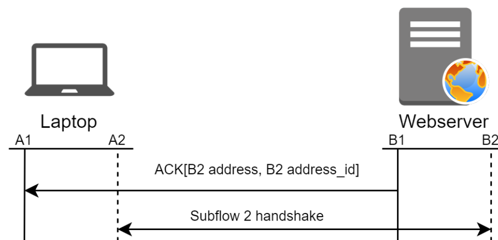


Figure 6: Sending an ADD_ADDRESS, resulting in a new subflow.

From the scenario's perspective: The webserver identifies that the laptop has an additional IP-address, A2, which was used to set up subflow 1. However, the laptop's address is positioned behind a NAT, implying that the webserver cannot set up an additional subflow with the newly identified IP-address and its own IP-address B2, because the NAT will reject the incoming connection attempt. To circumvent this, the webserver selects an established subflow (subflow 0) and sends, in this case, an ACK-packet carrying the ADD_ADDRESS option, containing the webserver's IP-address B2 and an identifier for B2. The laptop receives this packet and identifies the second IP-address of the webserver, B2. The laptop then decides to set up an additional subflow (subflow 2) on the address pair A2 and B2. See Figure 6 for a visualisation.

In the scenario both the webserver and the laptop attempted to create a subflow to add it to the existing connection. With MPTCP both endpoints are allowed to initiate new subflows for a given connection. However, an additional study [6] has provided the argument that, in practice, only the client is used to create new subflows, by directly initiating a new subflow with a server. This is because the clients often reside behind a NAT (or even a firewall) that prevents the server from successfully creating an additional subflow. It is still possible for the server to create subflows, when the client does not reside behind a NAT, but no precise requirements can be provided for allowing the server to do that.

By using this mechanism to exchange IP-address information between hosts, it is possible for a host to exchange a private IP-address in the ADD_ADDRESS option while being behind a NAT. For example, it could be that a storage device is connected to a local WiFi-network (IP-address S1), while also being connected with the Internet via 4G (IP-Address S2). The storage device may be connected to a remote phone via 4G (phone IP-address P2) and wishes to communicate to the phone that it also has a second IP-address, belonging to the local WIFI-network, available. The storage device does not know this, but the link between the WiFi-network and the Internet could be shielded by a NAT. Thus, the IP-address S1 that is being communicated could be a private address that cannot be connected to from the Internet.

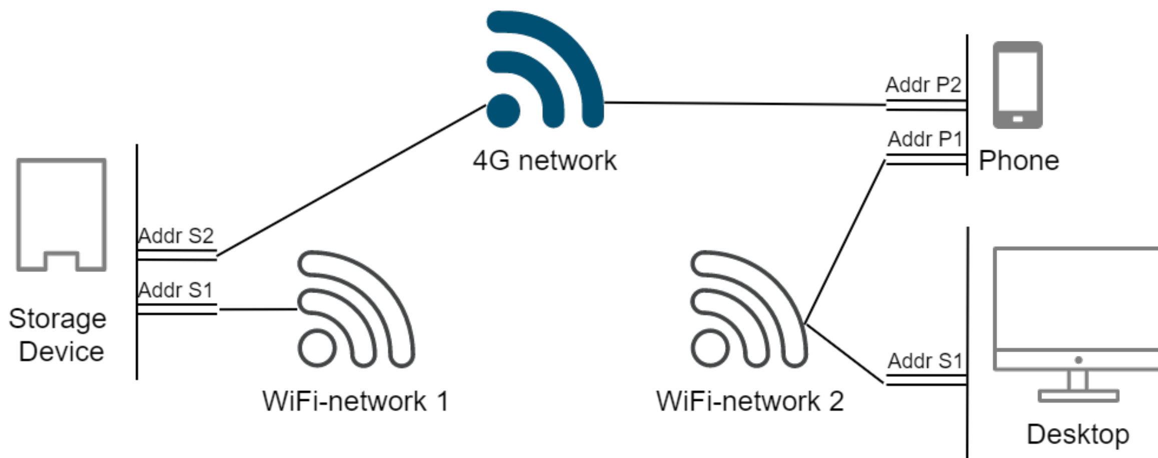


Figure 7: An example situation where two devices have the same address but are on different networks.

On top of that, the phone may also be connected to a different local WiFi-network and have a private IP-address P1. The phone may decide that it wants to create a new subflow on the address pair S1 and P1, which is allowed by the design of MPTCP. It could even be the case that a device, such as a desktop, with IP-address S1 exists on the WiFi-network that the phone is connected to, resulting in the phone initiating a connection attempt with that desktop, while it meant to initiate a connection with the storage device on a different WiFi-network. This connection attempt will fail, because the exchange of HMAC-messages prevents the phone from accidentally setting up a subflow with the desktop. This is because the HMAC-message of the desktop will be incorrect, which means that the phone will close the connection with that desktop. See Figure 7 for a network visualisation and see Figure 8 for a visualisation of the communication.

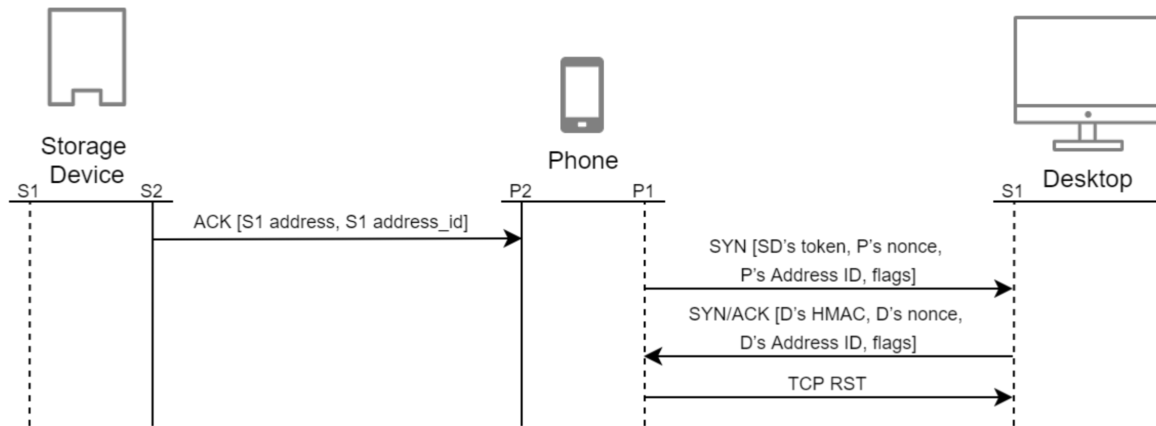


Figure 8: An example subflow initiation where the phone connects to the wrong device.

3.6 Removing an IP-address

In real world scenario's it is possible for an endpoint to lose its connection to a network. For example, a mobile phone that is connected to the WiFi-network of a user's home, as well as 4G. If the user decides to enter their car and drives away from their home, the connection between the phone and the WiFi's access point at home is lost, while the connection to 4G still remains. For a regular TCP connection this would mean that all TCP connections linked to the WiFi IP-address need to be terminated, as the phone's IP-address and port for the WiFi-network are no longer reachable. With the MultiPath extension the MultiPath-supporting connections would not need to be terminated immediately, because there may be other subflows in the connection that are coupled to IP-addresses that can still be reached by the peer, for example, a subflow could exist that is coupled to the IP-address for the phone's 4G connection.

With MultiPath an endpoint can inform their peer that one of its IP-addresses has become unavailable. This is done via the REMOVE_ADDRESS option when sending a packet. The option contains an address ID that identifies which IP-address needs to be removed, and consequently, which subflows need to be closed. However, it is possible that a malicious third party could be a part of the sequence of links in a path of one of the subflows.

This third party could modify a packet to carry the REMOVE_ADDRESS option, containing a potential valid address ID, which would then result in the receiver closing subflows that are linked to that address id. Thus, an endpoint stills need to check that the subflows, that use the mentioned IP-address, are no longer in use, preventing malicious packets from closing subflows.

From the scenario's perspective: IP-address A2 of the laptop is used to connect to a local WiFi-network, after downloading the large file for a while the laptop is moved to a different position that breaks the connection with the WiFi's access point, which results in the laptop losing IP-address A2. The laptop then informs the webserver of the loss of that IP-address by sending a packet over subflow 0 carrying the REMOVE_ADDRESS option, which contains the identifier of IP-address A2. The webserver receives this packet and checks if subflows 1 and 2 are no longer in use, because both subflows are coupled to IP-address A2. If the check completes, both subflows are closed by the webserver via a TCP RST and are also removed from the still existing connection with the laptop. Sending a TCP RST-packet allows for intermediate middleboxes to clean up state. See Figure 9 for a visualisation. After sending the REMOVE_ADDRESS option, the laptop could also initiate a new subflow between IP-addresses A1 and B2, to keep the connection in a state with more than one subflow, utilizing the mechanisms of MultiPath.

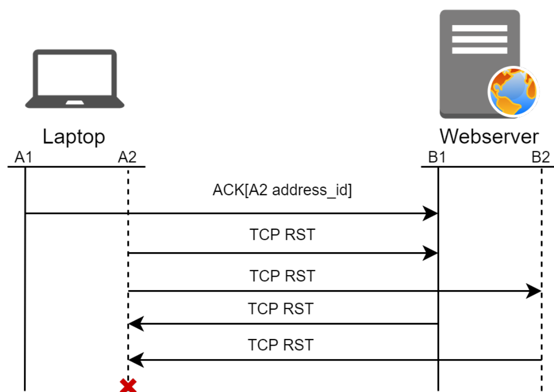


Figure 9: Sending a REMOVE_ADDRESS, resulting in closing subflows 1 and 2.

3.7 Transporting data

Once both endpoints have successfully set up multiple subflows within a single connection, the data that needs to be transported can be split over these subflows. The MultiPath extension provides a mechanism (section 3.7.1)to split a single input stream over the subflows, while also generating enough control information to reassemble the split data into a single, complete, and in-order output stream of data. By allowing a single input stream and generating a single output stream, the mechanism allows the extension to be transparent to the application layer of both the sending and a receiving endpoint. This means that already existing applications can utilize MultiPath without having to change their interaction with the TCP layer (the extension is also transparent to the lower layers).

An example of data transport via this mechanism in a MultiPath connection is given in section 3.7.2. However, there are some additional considerations that need to be taken when transporting data via this mechanism. This is because the use of multiple paths has an impact on the receiving capabilities of the receiver, and the transport capabilities of intermediate links in the network. A description about these considerations is given in section 3.7.3. Lastly, middlebox implementations can also affect the effectiveness of using multiple paths in a connection. Some middleboxes could, for example, rewrite the content of a packet, which could lead to mismatches when reassembling the data at the receiver side. A discussion about middlebox interference is given in section 3.7.4.

3.7.1 Mechanism to send data over multiple paths

In order to reassemble the split data received from multiple subflows, the receiving endpoint cannot solely rely on the sequence numbers (SNs) that are provided by TCP on each of the subflows. Take for example an endpoint with two subflows, subflow 0 and subflow 1. At a certain point in time the endpoint receives a packet that contains data on both subflows, and both packets contain the same sequence number X. The endpoint has no way of knowing if the data that was received on subflow 1 should be placed before the data that was received on subflow 0, or vice-versa. The endpoint does not even know if both pieces of received data should be placed one after the other, or if both parts occupy more distant locations in the stream, where other data that has not been received occupy the space in between. By only using the sequence numbers of the subflows the endpoint is placed in an ambiguous situation, where multiple outcomes are all equally valid. See Figure 10 for an example visualisation.

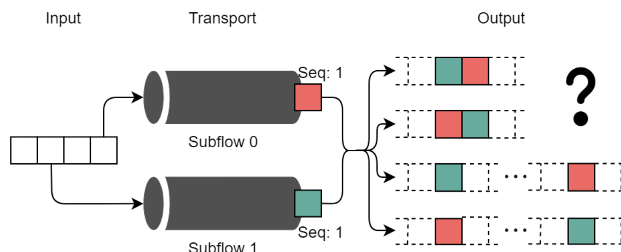


Figure 10: Reassembling packets from multiple subflows without data sequence number.

A new level of management must be introduced to solve this problem. MPTCP calls this the connection level, where it introduces a new data sequence space. A subflow still uses the normal TCP sequence and acknowledgement numbers to ensure that all its own data is delivered in a reliable and in-order manner between the endpoints, but the data sequence numbers (DSNs) from the data sequence space will dictate where all the subflow's data needs to be placed in the singular output stream on the receiving side. To communicate the data sequence numbers for each part of data from the sending endpoint to the receiving endpoint, the Data Sequence Signal option will be used. This option has two functionalities built into it. The first provides a mapping mechanism to link the sequence numbers of a subflow to data sequence numbers of the connection level. The second allows the acknowledgement of data on the connection level.

For the first functionality, a starting value is provided for both the TCP sequence number of a subflow and the corresponding data sequence number. In addition to this, a length value is provided as well, indicating how many bytes of transported data can be linked between the spaces with the current mapping. Using a length value for the mapping allows the sender to provide a mapping every so many packets that are sent over a specific subflow. One of the advantages that this approach has is that the overhead of the mapping can be kept to a minimum. Additionally, this approach was chosen instead of per-packet signalling, because the provided mapping is independent of the packet that carries it, due to the fact that the mapping is able to use a relative sequence number instead of the sequence number of the packet. This allows for middleboxes to perform segmentation and coalescing on the transmitted packets, where packets are combined in a new, larger packet, or where packets are split in multiple smaller packets. Lastly, this approach also allows multiple TCP sequence numbers from different subflows to be mapped to the same data sequence numbers, allowing a sender to retransmit/duplicate data over a different subflow.

Whenever a packet containing data is received by an endpoint, it is processed in two separate steps. In the first step the reception of the packet on a given subflow is acknowledged with the use of TCP ACKs, and the data is placed in a receive buffer. This buffer is used to hold packets from the sender for which the mapping has not yet arrived. At a later point in time the connection level takes these bits of split data from the buffer and assembles them back into a single output stream using the provided mappings in the Data Sequence Signal option. When the received data has been placed in the output stream, the receiving endpoint has to signal to the sender that the data has been received.

This is where the second functionality of the Data Sequence Signal option is used. This part of the option provides a DATA_ACK field that allows the acknowledgement of data on the connection level. Whenever a sending endpoint receives a TCP ACK and a DATA_ACK that acknowledges a certain part of data, it can safely assume that the receiver has received all the data in-order up to a certain point indicated by the DATA_ACK. This allows the sender to stop retransmitting the received data and move to the next part of the input stream. If the sender only received a TCP ACK for a certain part of data it needs to hold on to this data, because it may need to be retransmitted if it is declared as lost on the connection level. This can happen when, for example, the buffer at the receiving side is filling up with packets that contain mapped data, dropping packets that contain data for which the receiver has not yet received the mapping.

A sending endpoint is also able to inform a receiving endpoint that it has no more data to send. The sending endpoint can signal this with the use of the DATA_FIN field in the Data Sequence Signal option of a packet. From that point onward, the sender will only retransmit data that has not been acknowledged yet by a DATA_ACK. When all the data is successfully received by the receiving endpoint it will confirm this by sending a DATA_ACK that acknowledges the DATA_FIN signal. A connection is not closed unless there has been a DATA_FIN exchange or a timeout. This ensures that, when a DATA_FIN is sent, all the required data has been transmitted to the peer.

3.7.2 An example of data transport

Combining all this together, from the scenario's perspective: after performing all the previous operations, the laptop now has two subflows established in a connection with the remote webserver. The first subflow (subflow 0) is coupled to the IP-addresses A1 and B1. A second subflow (subflow 3) is coupled to the IP-addresses A1 and B2. The other two subflows (subflow 1 and 2) have been closed because the laptop has lost its WiFi-connection (and thus IP-address A2). An example transmission of "Hello world"-data can be seen in Figure 11. In this scenario the webserver decides to split "Hello world" into four parts: "He", "llo ", "wor" and "ld". The webserver decides to send the first part of the data to the laptop via subflow 0. In this packet the webserver provides a Data Sequence Signal option, containing a mapping between the sequence number of this packet (6000) and the connection-level data sequence number (0). In this case the data that is being sent is "He", which is the first part of the data. The length in this mapping also indicates that the mapping is only valid for the current packet. Upon receiving the packet on subflow 0 the laptop inspects the Data Sequence Signal option and concludes that the data that is being sent over subflow 0 should be placed at the beginning of the output stream.

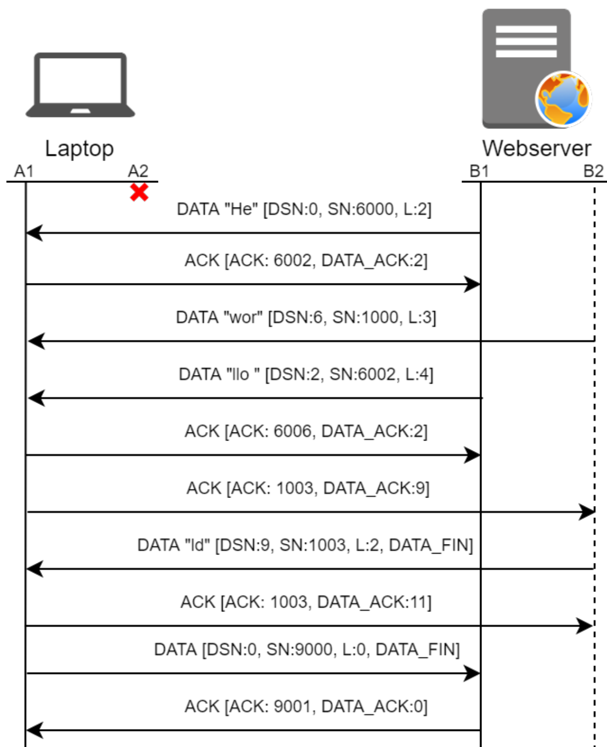


Figure 11: Transporting "Hello world" from the webserver to the laptop.

The laptop acknowledges the data by sending both a TCP ACK and a DATA_ACK back to the webserver over subflow 0. The TCP ACK tells the webserver that the laptop received the first packet that was sent on subflow 0, this is indicated by a value of 6002. The DATA_ACK tells the webserver that the laptop has placed the first two bytes of data into the output stream, this is indicated with a value of 2.

Upon receiving the TCP ACK and the DATA_ACK from the laptop over subflow 0, the webserver sees that the laptop has received the first packet that was sent over subflow 0. By inspecting the DATA_ACK the webserver identifies that the laptop has placed the first 2 bytes of the data in the output buffer, thus the webserver no longer needs to hold on to the first part of the data. The webserver then decides to send the third part of the data to the laptop via subflow 3. In this packet the webserver provides a Data Sequence Signal option, containing a mapping between the sequence number of this packet (1000) and the connection-level data sequence number (6). In this case the data that is being sent is "wor", which is the third part of the data. The length in this mapping also indicates that the mapping is only valid for the current packet. Immediately after sending the packet containing "wor" to the laptop, the webserver also decides to send the second part of the data to the laptop via subflow 0. In this packet the webserver provides a Data Sequence Signal option, containing a mapping between the sequence number of this packet (6002) and the connection-level data sequence number (2). In this case the data that is being sent is "llo ", which is the second part of the data. The length in this mapping also indicates that the mapping is only valid for the current packet.

The laptop received both packets at nearly the same time and places them in a buffer. The laptop now acknowledges the data that was sent on subflow 0 by sending both a TCP ACK and a DATA_ACK back to the webserver over subflow 0. The TCP ACK tells the webserver that the laptop received the second packet that was sent on subflow 0, this is indicated by a value of 6006. However, the DATA_ACK still has a value of 2, indicating that the laptop has not placed the data in the output stream yet.

The laptop then inspects its buffer and sees that the data that was received on subflow 3 ("wor") should be placed 6 bytes from the beginning of the output stream, which is 4 bytes after the data that was placed earlier, resulting in the following output stream: "He____wor". The laptop also sees that the data that was received on subflow 0 ("llo ") should be placed 2 bytes from the beginning of the output stream, which is in between the data that was placed earlier, resulting in the following output stream: "Hello wor". The laptop now acknowledges the data that was sent on subflow 3 by sending both a TCP ACK and a DATA_ACK back to the webserver over subflow 3. The TCP ACK tells the webserver that the laptop received the first packet that was sent on subflow 3, this is indicated by a value of 1003. However, due to the fact that the output stream no longer contains holes in the first 9 bytes, the DATA_ACK now has a value of 9, indicating that the laptop has placed all the previously sent data in the output stream.

Upon receiving the TCP ACK and the DATA_ACK from the laptop over subflow 0, the webserver sees that the laptop has received the second packet that was sent over subflow 0, but that the laptop has not yet placed the second and the third part of the data in the output stream. Thus, the webserver still needs to hold on to the data for potential retransmissions that could be required. Shortly afterwards the webserver receives the TCP ACK and the DATA_ACK from the laptop over subflow 3, and the webserver sees that the laptop also has received the first packet that was sent over subflow 3. By inspecting the DATA_ACK the webserver identifies that the laptop has placed the first 9 bytes of the data in the output buffer, thus the webserver no longer needs to hold on to that data.

The webserver then decides to send the last part of the data to the laptop via subflow 3. In this packet the webserver provides a Data Sequence Signal option, containing a mapping between the sequence number of this packet (1003) and the connection-level data sequence number (9). In this case the data that is being sent is "ld", which is the last part of the data. The length in this mapping also indicates that the mapping is only valid for the current packet. The webserver also signals that it has sent all the required data, by enabling the DATA_FIN flag in the option.

Upon receiving the packet on subflow 3 the laptop inspects the Data Sequence Signal option and concludes that the data that is being sent over subflow 3 should be placed at the end of the output stream. The laptop also sees that this packet completes the data that needed to be received by inspecting the DATA_FIN flag. The laptop acknowledges the data by sending both a TCP ACK and a DATA_ACK back to the webserver over subflow 3. The TCP ACK tells the webserver that it received the second packet that was sent on subflow 3, this is indicated by a value of 1005. The DATA_ACK tells the webserver that the laptop has placed the last two bytes of data into the output stream, this is indicated with a value of 11. At this point "Hello world" has been successfully transmitted between the webserver and the laptop. To complete the data transmission, the laptop also sends a packet over subflow 0 to the webserver to indicate that it has no more data to send. This is done by enabling the DATA_FIN flag in the option. Finally, upon receiving this packet the webserver sees that this packet completes the data that needs to be transmitted, and acknowledges this last packet by sending both a TCP ACK and a DATA_ACK back to the webserver over subflow 0.

3.7.3 Flow control and congestion control

We see that with the use of the Data Sequence Signal the sender can transmit data to a receiver over multiple subflows during a single connection. However, like a regular TCP connection, a sending endpoint also needs to consider that the receiving endpoint may not be able to receive the data at the same rate that the sender is transmitting. A regular TCP connection advertises a receive window in the headers of each packet, which tells the sender how much data the receiver is willing to accept past the ACKed amount. MultiPath also advertises a connection-level receive window, which is shared among the subflows and is relative to the DATA_ACK on the connection level. This shared receive window signals to the sending endpoint how much data the receiver can accept on the subflows from the indicated point in the stream. A shared receive window was chosen over a per-subflow receive window, because a shared receive window allows for any subflow to send data that fits in the window, whereas a per-subflow receive window could lead to underutilization if some subflows would not use up their window.

With this mechanism, all subflows should advertise the same size for the shared receive window. However, it is possible that middleboxes on a subflow's path have altered that value. This happens when, for example, one of the middleboxes on that path does not have enough buffering for all the data to be propagated to the receiver. This can result in multiple different values being advertised among the subflows. To overcome this the sender only uses the largest recently advertised receive window as the actual shared receive window. However, when sending data over a subflow that has a smaller advertised window the sender is only allowed to send data that fits within that subflow's window.

Only using a shared advertised receive window is not enough. The receive window solely prevents the sender from overloading the receiver, but it does not prevent the server from overloading the network. With regular TCP a congestion window (that is not communicated to the peer), in combination with a congestion controller, is used to prevent/handle congestion on the network. In MPTCP the basic idea of a congestion window is also applied. Each subflow has its own congestion window in which it is allowed to send data to the receiver, this congestion window will grow and shrink based on the level of congestion on the subflow's path. However, just implementing a separate congestion controller for each subflow would lead to unfairness with other (regular) TCP connections if the current existing MultiPath connection utilizes multiple paths.

In MultiPath, the congestion window of each subflow is coupled together. An initial congestion control scheme, called LIA [7], was proposed for use with MPTCP. The scheme stated three goals that need to be fulfilled by a MultiPath congestion controller in order to be desirable. The first goal was to improve throughput, where a MultiPath connection should perform at least as well as a regular TCP connection. A second goal states that the congestion controller should do no harm, by not taking up more capacity from a resource than a regular TCP connection would. The third goal aims for traffic redirection, where a MultiPath connection should, with regard to the first two goals, move as much traffic as possible away from the most congested paths.

The proposed scheme did not completely fulfill the third goal, as discussed in its RFC. Other congestion control schemes have been proposed for MPTCP, such as OLIA [8], BALIA [9], wVegas [10] and mpCUBIC [11]. These schemes measure congestion on each subflow and try to avoid using those with the most congestion. The congestion windows of the subflows are manipulated in a way specific to the scheme and are based on the level of congestion. Of the previously mentioned schemes, OLIA is implemented and used in the MPTCP custom Linux kernel implementation [12]. A more in depth discussion about flow control and congestion control in a MultiPath connection can be found in section 8.1.

3.7.4 Middlebox interference

With the use of an advertised receive window and (non-advertised) congestion windows, the sender is able to transmit data to a receiver over multiple subflows during a single connection, where the sender applies flow control to avoid overloading the transmission capabilities of the paths and the receiving capabilities of the receiver. However, there are still some problems that can occur while transporting data over multiple subflows.

For example, a middlebox implementation could be operating in such a way that it only allows additional TCP options to be part of a packet when a new connection is being initialized, thus stripping all unknown options once a connection has been established. For a MultiPath connection this would mean that all Data Sequence Signal options could be stripped from the packets that are sent over a subflow, resulting in the receiver being unable to place the data in the output stream, because no mapping information is being provided.

The receiver handles this problem by performing a fallback operation where it closes the subflow that delivers stripped packets if no mapping has been provided after a set amount of time. The sender can also close a subflow if no Data Sequence Signal options are carried by the packets it receives over that subflow. If this problem occurs on the initial subflow over which the connection has been established, while no other subflows have been established, a different solution needs to be used. In this case the sender can provide an infinite mapping for the data, and both endpoints can perform a fallback to a regular TCP connection.

It is also possible for middleboxes to alter the payload of a packet. For example, many NAT devices include functionality for protocols such as FTP where private IP-addresses in the FTP control channel are re-written to the correct public IP-address imposed by the NAT. These address changes can lead to a difference in the length of the payload, which can break the mapping that is provided in the Data Sequence Signal option, resulting in data being placed incorrectly on the output stream [13]. Via checksums, of which the use is signalled during connection initialization, a receiver is able to detect if the payload of a received packet has been altered. (TCP also has this problem of content rewriting. However, the middlebox that alters the content of the packet also alter the sequence numbers, thus no further problems arise. In MPTCP this is a problem because the Data Sequence Signal option is not altered to reflect this change.)

If this is the case, the receiver can perform a fallback operation based on the current state of the connection. If other subflows still exist within the connection, the current subflow can be closed by sending a TCP RST packet, carrying the MP_FAIL option, containing the data sequence number from the start of the segment that had the checksum failure. However, if no other subflows exist the receiver can fall back to a regular TCP connection by sending an ACK that carries the MP_FAIL option. The sender will identify this fallback to a regular TCP connection if it no longer receives any MultiPath signalling and will, in response, also fall back to a regular TCP connection after providing an infinite mapping for the data.

Explaining the issue from the scenario's perspective: The webserver also sends the data "Hello world" to the laptop (see Figure 12). However, this time checksums are enabled. The webserver decides to split the data into four parts: "He", "llo ", "wor" and "ld". It then decides to send the first part of the data to the laptop via subflow 0. In this packet the webserver provides a Data Sequence Signal option, containing a mapping between the sequence number of this packet (6000), the connection-level data sequence number (0) and a checksum (0x53). In this case the data that is being sent is "He", which is the first part of the data. The length in this mapping indicates that the mapping is valid for the next couple of packets.

Upon receiving the packet on subflow 0 the laptop inspects the Data Sequence Signal option and concludes that the data that is being sent over subflow 0 should be placed at the beginning of the output stream. The laptop also stores the checksum. The laptop acknowledges the data by sending both a TCP ACK and a DATA_ACK back to the webserver over subflow 0. The TCP ACK tells the webserver that the laptop has received the first packet that was sent on subflow 0, this is indicated by a value of 6002. The DATA_ACK tells the webserver that the laptop has placed the first two bytes of data into the output stream, this is indicated with a value of 2. A checksum is also sent back to the webserver (0xaa).

Upon receiving the TCP ACK and the DATA_ACK from the laptop over subflow 0, the webserver validates the checksum and sees that the laptop has received the first packet that was sent over subflow 0. By inspecting the DATA_ACK the webserver identifies that the laptop has placed the first two bytes of the data in the output buffer, thus the webserver no longer needs to hold on to the first part of the data.

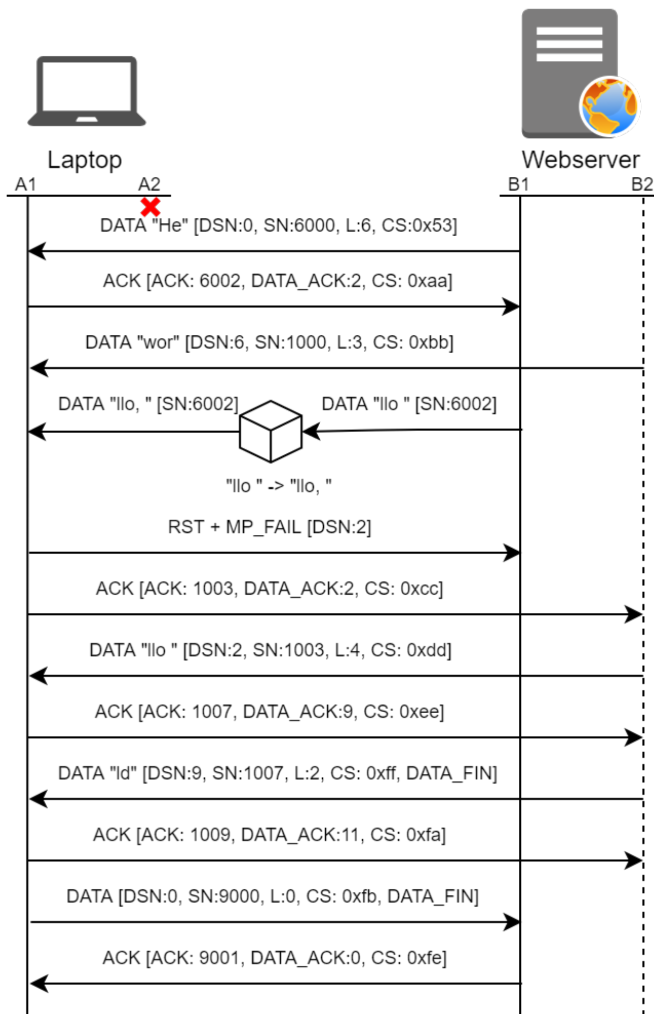


Figure 12: An error occurs while transporting Hello world from the webserver to the laptop.

The webserver decides to send the third part of the data to the laptop via subflow 3. In this packet the webserver provides a Data Sequence Signal option, containing a mapping between the sequence number of this packet (1000) the connection-level data sequence number (6), and a checksum (0xbb). In this case the data that is being sent is “wor”, which is the third part of the data. The length in this mapping also indicates that the mapping is only valid for the current packet. Immediately after sending the packet containing “wor” to the laptop, the webserver also decides to send the second part of the data to the laptop via subflow 0. In this packet the webserver does not provide a Data Sequence Signal option, because the data that is being sent is already mapped by the previous packet that was sent on subflow 0. In this case the data that is being sent is “llo ”, which is the second part of the data.

The laptop received both packets at nearly the same time and places them in a buffer. However, the packet that was received on subflow 0 has been altered by a middlebox on the path, instead of "llo " the packet now contains "llo, ". Upon validating the checksum (0x53) that was received with the first packet on subflow 0 the laptop identifies that the data has been altered. Because there is a second subflow active in the connection the laptop decides to close the subflow by sending back a TCP RST packet to the webserver, carrying the MP_FAIL option, containing the data sequence number 2. This tells the webserver that there was a problem when the laptop was trying to reconstruct the data, and that subflow 0 needs to be removed from the connection. The laptop then inspects its buffer and sees that the data that was received on subflow 3 ("wor") should be placed 6 bytes from the beginning of the output stream, which is 4 bytes after the data that was placed earlier, resulting in the following output stream: "He____wor".

The laptop now acknowledges the data that was sent on subflow 3 by sending both a TCP ACK and a DATA_ACK back to the webserver over subflow 3. The TCP ACK tells the webserver that the laptop received the first packet that was sent on subflow 3, this is indicated by a value of 1003. However, due to the fact that a part of the data that was sent over subflow 0 was corrupted the output stream still contains holes in the first 9 bytes. Thus, the DATA_ACK still has a value of 2, indicating that the laptop has only placed the first two bytes of the data in the output stream. A checksum is also sent back to the webserver (0xcc). Upon receiving the TCP RST from the laptop over subflow 0, the webserver identifies that the second data part, containing "llo " was not accepted by the laptop, because the data sequence number in the MP_FAIL option has a value of 2. The webserver closes the subflow and reschedules the data part to be sent over subflow 3. Luckily, the next packet that can be sent over subflow 3 is not part of any existing mapping, thus "llo " can be sent as the next packet on subflow 3. Shortly afterwards the webserver receives the TCP ACK and the DATA_ACK from the laptop over subflow 3, the webserver validates the checksum and sees that the laptop has received the first packet that was sent over subflow 3. By inspecting the DATA_ACK the webserver identifies that the laptop has only placed the first two bytes of the data in the output buffer.

The webserver then decides to send the rescheduled part of the data to the laptop via subflow 3. In this packet the webserver provides a Data Sequence Signal option, containing a mapping between the sequence number of this packet (1003), the connection-level data sequence number (2) and a checksum (0xdd). In this case the data that is being retransmitted is "llo ", which is the third part of the data. The length in this mapping also indicates that the mapping is only valid for the current packet. Upon receiving the packet on subflow 3 the laptop validates the checksum and inspects the Data Sequence Signal option. It concludes that the data that is being sent over subflow 3 should be placed 2 bytes from the beginning of the output stream, which is in between the data that was placed earlier, resulting in the following output stream: "Hello wor". The laptop now acknowledges the data that was sent on subflow 3 by sending both a TCP ACK and a DATA_ACK back to the webserver over subflow 3. The TCP ACK tells the webserver that the laptop received the second packet that was sent on subflow 3, this is indicated by a value of 1007. However, due to the fact that the output stream no longer contains holes in the first 9 bytes, the DATA_ACK now has a value of 9, indicating that the laptop has placed all the previously sent data in the output stream.

The webserver receives the TCP ACK and the DATA_ACK from the laptop over subflow 0, the webserver validates the checksum and sees that the laptop has received the second packet that was sent over subflow 0. By inspecting the DATA_ACK the webserver identifies that the laptop has placed the first nine bytes of the data in the output buffer, thus the webserver no longer needs to hold on to the first three parts of the data. The webserver then decides to send the last part of the data to the laptop via subflow 3. In this packet the webserver provides a Data Sequence Signal option, containing a mapping between the sequence number of this packet (1007), the connection-level data sequence number (9) and a checksum (0xff). In this case the data that is being sent is "ld", which is the last part of the data. The length in this mapping also indicates that the mapping is only valid for the current packet. The webserver also signals that it has sent all the required data, by enabling the DATA_FIN flag in the option.

Upon receiving the packet on subflow 3 the laptop validates the checksum and inspects the Data Sequence Signal option. It concludes that the data that is being sent over subflow 3 should be placed at the end of the output stream. The laptop also sees that this packet completes the data that needed to be received by inspecting the DATA_FIN flag. The laptop acknowledges the data by sending both a TCP ACK and a DATA_ACK back to the webserver over subflow 3.

The TCP ACK tells the webserver that it received the third packet that was sent on subflow 3. This is indicated by a value of 1009. The DATA_ACK tells the webserver that the laptop has placed the last two bytes of data into the output stream. This is indicated with a value of 11. At this point "Hello world" has been successfully transmitted between the webserver and the laptop. To complete the data transmission, the laptop also sends a packet over subflow 0 to the webserver to indicate that it has no more data to send. This is done by enabling the DATA_FIN flag in the option. Finally, upon receiving this packet the webserver sees that this packet completes the data that needs to be transmitted, and acknowledges this last packet by sending both a TCP ACK and a DATA_ACK back to the webserver over subflow 0.

3.8 Closing an MPTCP connection

On a regular TCP connection a sender can signal to the receiver that it has sent all data; this is done by using the FIN flag in the TCP header of a packet. As previously mentioned, with MultiPath TCP an additional DATA_FIN flag was introduced to signal that all data has been sent on the connection level. Whenever the sender signals a DATA_FIN it must wait for that packet to be acknowledged by the receiver with a DATA_ACK before it is able to start closing the connection. A receiver will only acknowledge this DATA_FIN once all data has been successfully reassembled at the connection level.

If the DATA_FIN has been acknowledged by a DATA_ACK both endpoints can start closing the connection by exchanging packets containing the TCP FIN flag on each individual subflow. This helps middleboxes along the path to clean state. A subflow is only considered closed once both endpoints acknowledged each other's FIN with a TCP ACK (or when a timeout has occurred). A connection is considered closed once both endpoints have acknowledged each other's DATA_FIN with a DATA_ACK.

From the scenario's perspective: The webserver has already successfully sent the first three packets containing "He", "llo " and "wor" to the laptop, as can already be seen in Figure 11. The webserver now sends the last packet, containing "ld" from "hello world", by enabling the DATA_FIN flag in the Data Sequence Signal. The laptop sees this DATA_FIN flag and acknowledges the packet with a DATA_ACK, because it has also received all the previous packets and was able to place "hello world" in the output buffer. Afterwards, the laptop also sends a packet over subflow 3 to the webserver to indicate that it has no more data to send. This is done by enabling the DATA_FIN flag in the Data Sequence Signal option. Finally, upon receiving this packet the webserver sees that this packet completes the data that needs to be transmitted, and acknowledges this last packet by sending both a TCP ACK and a DATA_ACK back to the webserver over subflow 3.

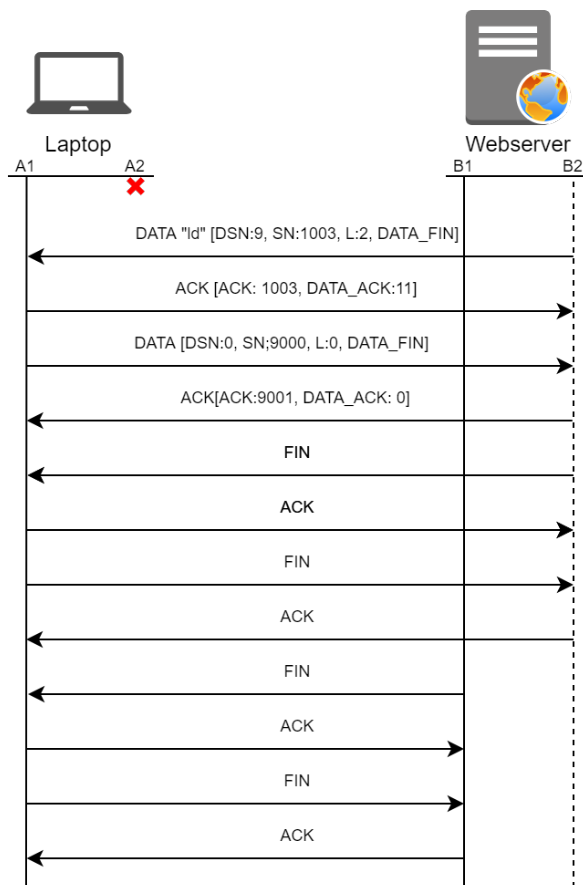


Figure 13: Closing a MPTCP connection.

At this point, when the laptop has received the webserver's DATA_ACK, both endpoints have confirmed that all data has been sent and that the connection can be closed. This is done by both endpoints sending a packet, containing an enabled FIN-flag, to the peer over each subflow, being subflow 1 (from A1 to B1) and subflow 3 (from A1 to B2). Both endpoints also acknowledge those packets that contain the enabled FIN flag. A visual representation can be seen in Figure 13.

It is also possible that an endpoint wants to abruptly close a connection when, for example, the endpoint is running out of system resources. In regular TCP this is done by using the RST flag in the TCP header of a packet, indicating that the host will no longer accept data. In MPTCP this RST flag is used to abruptly close the subflow over which the flag was sent. The other subflows that are part of the connection will not be closed, and the connection will continue to exist. Therefore, MPTCP introduces the MP_FASTCLOSE option to abruptly close a connection.

An endpoint can do this by sending a packet carrying the MP_FASTCLOSE option, containing the peer's key over one subflow, while also sending packets containing the TCP RST flag over all the other subflows. The host can only close the entire connection, along with the remaining subflow if it receives a TCP RST from its peer over that subflow. By sending the peer's key a layer of protection is implemented against malicious third parties on a path that could add the MP_FASTCLOSE option to a packet. These third parties do not know the key that needs to be transmitted, because the keys have only been exchanged once by both endpoints during the connection initialization.

By allowing the use of both a regular RST and a MP_FASTCLOSE, the extension permits "break-before-make" scenarios where connectivity is lost on all subflows before a new one can be re-established. In this case the connection remains alive for a predefined amount of time where both endpoints can try to establish a new subflow to continue the exchange of data.

3.9 Problems of MPTCP

While some problems that can occur during a MultiPath TCP connection are being handled by the mechanisms in place, an additional problem could also occur that has not been discussed before. The TCP options field in the TCP header of a packet has limited space of 40 bytes, due to the data offset field having a maximum value of 60, indicating how many additional bytes are used for flags, checksum, urgent pointer and options before the actual data starts. An endpoint that has multiple TCP extensions enabled will have to share that options space between these extensions. It is possible that there is not enough space left to add an additional MPTCP signal to the packet when signals from other extensions have already been added. To overcome this, an endpoint can send a duplicate ACK that carries these additional MPTCP signals. Meaning that, if an MPTCP endpoint receives a duplicate ACK, it should not be used as an indication of possible congestion, whereas this would be the case in regular TCP. If no duplicate ACK can be sent, no MultiPath options can be sent over that subflow. Thus, a fallback operation is performed where the connection either closes the subflow, of reverts the connection back to a regular TCP connection.

Sending a duplicate ACK to communicate MultiPath options that did not fit in the options-field of a previous packet is not the only change that was made to the semantics of TCP. By adding an additional management layer, called the connection layer, MultiPath also changes the semantics of the RST and FIN-flags, by only letting them influence the subflow that it was sent on. As mentioned in the previous section, the extension added MP_FASTCLOSE and DATA_FIN flags that influence the entire connection. Additionally, the acknowledgement mechanism of TCP was also altered by introducing DATA_ACKs.

With the many different middlebox implementations of TCP that can perform actions which could break a connection unexpectedly, MultiPath TCP was designed with the idea to provide a best effort to make a MultiPath connection work. A MultiPath connection should always be attempted, and when problems arise the connection should still be kept alive, even if it means falling back to a regular TCP connection.

3.10 MPTCP deployment

MultiPath TCP has seen additional research since its RFC came out in January 2013. For example, a series of additional congestion controllers were proposed to enhance the perfomance of a MPTCP connection (see section 3.7.3). Only a couple of months after the publication, Apple implemented and enabled MultiPath TCP on all its iOS devices. Since then, MultiPath TCP has also been implemented on systems that run a custom Linux kernel from UCLouvain, Apple MacOS, FreeBSD Solaris and others [14, 15]. There is still an ongoing effort to implement MPTCP in the main Linux kernel [16]. In 2015 a study [17] was performed on MultiPath internet traffic that was captured on a single server. Here, 190,451 MPTCP connections that originated from more than 7,000 different hosts were analysed. The paper makes interesting observations. For example, very few fallback operations caused by middleboxes were performed. Additionally, 86% of the connections established an additional path immediately to send data. The remaining 14% only used MultiPath to have resistance against network failures (where the connection had to be closed if regular TCP was used).

4 Quick UDP Internet Connections (QUIC)

One of the main problems in the current design of TCP is the fact that connections between two endpoints can suffer from HoL-blocking when the connection experiences packet drops. From the perspective of TCP, the data that is transmitted to the peer is a single large bytestream that has to be delivered in an in-order fashion to ensure that the receiving endpoint can see the data as intended. If a packet is dropped, there is no other option than to retransmit that same packet to the peer, halting the transmissions of all other remaining packets. HTTP/2 aimed to solve this problem by introducing streams, where data from multiple files could be multiplexed and placed in packets for TCP to transport to the remote peer. Via this approach, it would still be possible for a packet to be dropped on the wire. However, if this packet contained data from a single stream, the data transmission from the other streams would not have to be halted. There was still one problem with this approach, because TCP was not aware of these additional streams. A first solution would be to change the protocol design of TCP to make it stream-aware. However, it was impossible to change TCP itself to do so [18]. Thus, a decision was made to create a new transport protocol in the form of QUIC, that is built on top of UDP, and implements its own mechanisms to provide a reliable transport of data via multiple streams.

Besides solving the HoL-blocking problem, the design of QUIC also aimed to solve additional issues with the current design of TCP. For example, mechanisms were introduced to enable, but not limit the design to, implementing security and encryption for header values, allowing the connection to migrate to different 4-tuples and reducing connection setup time. To keep a fixed focus on the design of QUIC, a working group was established, where discussions about the mechanisms and design visions can take place. In general, the QUIC protocol is a bundling of layers where mechanisms are tightly coupled together to support the safe and reliable transport of data. Within these mechanisms, a focus is placed on providing security for the data that needs to be transported. For example, almost everything that is placed in a packet is encrypted, even most of the packet header. Additionally, the protocol also took the idea of providing multiple data streams, and implemented it at a transport level to enable better data transport (making QUIC stream aware). There are also other aspects implemented in QUIC, such as the implementation of congestion control and opening new connections in 0-RTT. Since the focus of this thesis is the implementation of MultiPath in QUIC, only the important mechanisms for this feature are explained, being the use of connection identifiers (section 4.1), and the ability to migrate an established connection (section 4.2).

4.1 Connection identifiers

In TCP, a connection is seen as a single path connecting two endpoints to each other. For simplicity's sake, this connection had to be identified by both endpoints via information that is available to the endpoints. Therefore, the 4-tuple existing of the IP-address (and port) that each host uses to send data to the peer was a good candidate. Both hosts use the IP-addresses from the packets to identify to which connection the packets belong to. However, by identifying a connection via a 4-tuple of IP-addresses, problems arise when either endpoint loses its access to that IP-address. Therefore, in QUIC a different approach is taken where connections are no longer identified via IP-address information.

Instead of using IP addresses, a different concept, called connection identifiers (IDs) is used. An endpoint is able to generate a unique set of connection IDs that it will use to identify a given connection. Thus, whenever a packet arrives at the endpoint, it will be marked with one of the connection IDs that the endpoint has chosen. In order to make this work, the endpoint has to communicate these connection IDs to its peer, which in turn will use them to mark packets that are destined for the endpoint. The communication of connection IDs can be done via two methods. The first one being the use of initial packets that are sent to the peer to establish a connection. The second method takes place when a connection has already been established. In the first method, when an endpoint wants to initiate a connection with a remote peer, it will mark the initial packet with a source connection ID that the endpoint has chosen. A random destination connection ID is also given to the packet as well. After this, the endpoint sends the packet to the remote peer. The peer will process the packet and extract the source connection ID that the first endpoint has chosen. It will store this value as a connection ID that must be used when sending packets back to the endpoint. In response, the peer will also send an initial packet to the endpoint. This time, the source connection ID carries a value that the peer has chosen to identify the connection. Additionally, the packet is also given a destination connection ID value. More specifically, this value is the same as the connection ID that the peer previously stored.

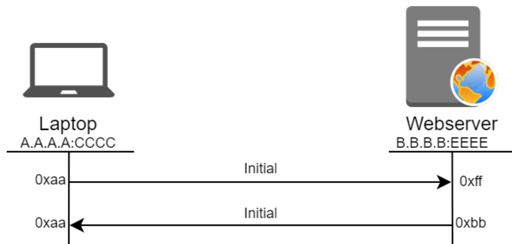


Figure 14: A simplified QUIC connection setup.

A given scenario: a modern laptop wants to connect to a remote webserver. In its first packet, the laptop places a source connection ID value of 0xaa, while the destination connection ID is given a value of 0xff. This means that the laptop has generated the connection ID (0xaa) to identify the connection with the remote webserver. After this, the laptop sends the packet to the webserver, that notices upon reception that the laptop is trying to initiate a connection. The webserver extracts the source connection ID from the packet (0xaa), and stores it. The webserver now knows that the laptop will be able to identify the connection if it sends future packets to the laptop that are marked with this connection ID. In response to the packet from the laptop, the webserver generates an initial packet. This packet is given a source connection ID value of 0xbb, while the destination connection ID is given a value of 0xaa. This means that the webserver has generated the connection ID (0xbb) to identify the connection with the laptop. After this, the webserver sends the packet to the laptop. In the last step, the laptop receives the packet and checks the destination connection ID. Here, the laptop sees that the value (0xaa) matches the connection ID that the laptop associated to the connection with the webserver. The laptop extracts the source connection ID from the packet (0xbb), and stores it as well. At this point, both endpoints have received a connection ID from their peer that they will use to mark packets belonging to that connection. See Figure 14 for a visualisation.

The second method to exchange additional connection IDs is used when the connection has already been established. Here, a NEW_CONNECTION_ID-frame is used to communicate the connection IDs to the peer. From the scenario's perspective: after establishing a connection with the remote webserver, the laptop decides to provide the webserver with an additional connection ID. Thus, the laptop generates another connection ID (0xaf) and places it in a NEW_CONNECTION_ID-frame. The laptop then places the frame in a 1RTT packet and marks the packet with the webserver's connection ID (0xbb). Finally, the laptop sends the packet to the webserver.

The webserver receives the packet and reads the destination connection ID (0xbb). The webserver then identifies that the packet belongs to the connection with the laptop. After this, the webserver decrypts the packet and reads the content of the NEW_CONNECTION_ID-frame. Here, it stores the laptop's additional connection ID (0xaf). Finally, the webserver sends back a 1RTT-packet to the laptop carrying an ACK-frame to acknowledge the reception of the packet sent by the laptop. This packet also carries the destination connection ID of the laptop (0xaa). See Figure 15 for a visualisation.

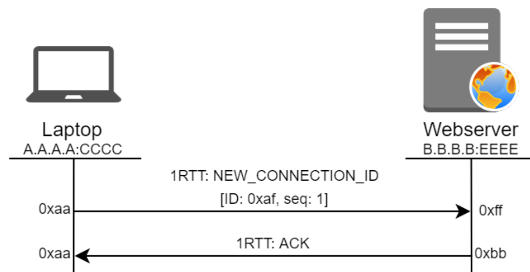


Figure 15: A new connection ID being exchanged.

At this point, the webserver has two connection IDs that can be used to mark packets destined for the laptop. An important note to make here is that the laptop must keep track of the connection IDs that it previously communicated to the webserver. This is to ensure that the laptop can keep identifying validly marked packets that carry any of the previously communicated connection IDs. Additionally, the webserver now also has the option to choose to no longer use one of these connection IDs. Here, a RETIRE_CONNECTION_ID-frame can be used to communicate the removal of the connection ID.

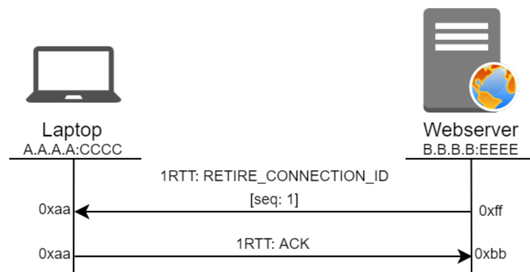


Figure 16: A connection ID being retired.

From the scenario's perspective: after a while, the remote webserver decides to no longer use the second connection ID (0xaf) that it received from the laptop. Thus, the webserver generates a RETIRE_CONNECTION_ID-frame and places the connection ID's sequence number (that was previously received via the NEW_CONNECTION_ID-frame) in it. The webserver then places the frame in a 1RTT packet and marks the packet with the laptop's connection ID (0xaa). Finally, the webserver sends the packet to the laptop.

The laptop receives the packet and reads the destination connection ID (0xaa). The laptop then identifies that the packet belongs to the connection with the remote webserver. After this, the laptop decrypts the packet and reads the content of the RETIRE_CONNECTION_ID-frame. Here, it identifies that the connection ID that it generated earlier (0xaf), will no longer be used by the webserver to mark packets. Thus, the laptop removes the mentioned connection ID. Finally, the laptop sends back a 1RTT-packet to the webserver carrying an ACK-frame to acknowledge the reception of the packet that the webserver has sent. This packet also carries the destination connection ID of the webserver (0xbb). See Figure 16 for a visualisation.

While the connections are now identified via connection IDs that each endpoint has chosen, the IP-addresses from which packets are sent are still being tracked by both endpoints. The connection between two endpoints is temporarily bound to a given 4-tuple. The word "temporarily" is used here because the QUIC connection is also able to migrate to a different 4-tuple during its lifetime.

4.2 Connection migration

While both endpoints are able to identify the connection via the connection IDs that they have exchanged, a new possibility has opened up for the other mechanisms of QUIC to enable. As mentioned in the previous section, a connection is temporarily bound to a given 4-tuple of IP-addresses from both endpoints. Inherently, the design of QUIC currently only allows for a single path to be used to transport data between two endpoints. However, because the connection no longer directly requires path-based information (such as IP-addresses) to be identified, a connection can be migrated to different paths (thus, different 4-tuples) that also connect the two endpoints. This is called connection migration.

Modern systems often have multiple NICs installed, giving those systems multiple IP-addresses that can appear and disappear at any time. A mobile phone can, for example, be connected to both a WiFi-connection and a 4G-connection. Here, the mobile phone could connect to a remote media server by initiating the connection via its WiFi IP-address. While remaining connected to this media server, the phone might be moved to a location that is out of the WiFi-signal's reach. Under normal circumstances, and with a regular TCP connection, the connection must be closed, since the phone no longer has access to the IP-address from the WiFi-signal. In contrast, in a QUIC connection the mobile phone still has its 4G-connection established, and thus it has another IP-address that it can use to send data to the media server. Since the connection is identified via connection IDs that were previously exchanged, the mobile phone can still send packets to the media server, as long as those packets are marked with connection IDs that the media server can use to correctly identify the connection.

However, this transition to a new IP-address cannot just happen immediately, there are a number of steps that need to be taken to ensure that the connection can proceed without future problems. In the previous section it is mentioned that an endpoint could use any of the communicated connection IDs to mark packets. However, a better method is to keep using the same connection ID for as long as possible, while the connection remains bound to the same 4-tuple. In the first step, if an endpoint initiates migration to a different IP-address, it should switch to using a different connection ID. If a migration occurs and the same connection IDs are still used to mark packets (in both directions), malicious third parties could link together the different IP-addresses that were used to send the same connection ID, and the location of the endpoint could become compromised. Thus, during the lifetime of a connection, both endpoints should provide their peers wit additional connection IDs to allow for a connection to migrate safely. The second step that needs to be taken is that a migrating endpoint should first check if it can reach the peer via the new 4-tuple, before fully committing the connection to the new 4-tuple. For this, the QUIC protocol provides a path validation procedure. Here, a migrating endpoint generates a challenge value that is placed inside a PATH_CHALLENGE-frame. The frame is then sent over the new 4-tuple towards the peer via a packet. If the peer is able to respond via the same 4-tuple with a packet carrying a PATH_RESPONSE-frame containing the same challenge value, the path validation procedure completes successfully and the endpoint is allowed to migrate. In response to detecting the migration, the peer is also allowed (and encouraged) to perform its own path validation procedure to check for connectivity.

From the scenario's perspective: assuming that the connection between the laptop and the webserver has been established for a while, both endpoints have exchanged additional connection IDs with each other. The laptop has given the webserver the connection ID (0xab), while the webserver has given the laptop the connection ID (0xef). Also assume that the connection between the laptop and the webserver has been temporarily bound to the 4-tuple (A.A.A.A:CCCC, B.B.B.B:EEEE). At some point during the connection, the laptop discovers that it has lost its initial IP-address, but that it also has an additional (second) IP-address available, being D.D.D.D:FFFF. The laptop decides to attempt a connection migration to bind the connection to this additional IP-address. Here, the laptop generates a random challenge value (0xabc) and places it in a PATH_CHALLENGE-frame. After this, the laptop places the frame in a 1RTT-packet and marks it with a different connection ID than that it used before (0xef). Lastly, the laptop sends the packet to the webserver via its second IP-address.

The webserver receives the packet and reads the destination connection ID (0xef). The webserver then identifies that the packet belongs to the connection with the laptop. Here, the webserver notices that a different connection ID was used by the laptop, and that the source IP-address has not been seen before. The webserver now presumes that the laptop is actively trying to migrate the connection to the new IP-address. The webserver decrypts the packet and reads the content of the PATH_CHALLENGE-frame. Here, it extracts the challenge value (0xabc), and immediately places the value in a PATH_RESPONSE-frame. After this, the webserver also generates a random challenge value (0xdef), because it also wants to initiate a path validation procedure to check connectivity with the laptop. This challenge value is then placed in a PATH_CHALLENGE-frame.

Next, the webserver places each frame in a different 1RTT packet, and marks both packets with a different connection ID that has not been used before (0xab). Lastly, the webserver sends the packets to the laptop, destined to the new IP-address of the laptop.

The laptop receives both packets on its second IP-address, and reads the destination connection IDs (both 0xab). The laptop then identifies that the packet belongs to the connection with the webserver. The laptop notices that the webserver also has used a different connection ID. The laptop decrypts the packets and reads the contents of the PATH_RESPONSE-frame and the PATH_CHALLENGE-frame. The value from the first frame (0xabc) is then compared to the value that the laptop had generated earlier and notices that the two values match. Thus, the laptop can conclude that the path validation procedure has completed successfully, resulting in the laptop binding the connection to the new 4-tuple (D.D.D.D:FFFF, B.B.B.B:EEEE). After this, the laptop places the challenge value from the second frame (0xdef) in a PATH_RESPONSE-frame. The laptop then places the frame in a 1RTT-packet and marks it with the webserver's connection ID (0xef). Finally, the laptop sends the packet to the webserver via its second IP-address.

The webserver receives the packet and reads the destination connection ID (0xef). The webserver then identifies that the packet belongs to the connection with the laptop. Here, the webserver decrypts the packet and reads the content of the PATH_CHALLENGE-frame. The webserver extracts the challenge value, and compares it to the value that it had generated earlier. The webserver notices that the two values match, and can conclude that the path validation procedure has completed successfully. The webserver then also binds the connection to the new 4-tuple. At this point, both endpoints have performed the path validation procedure and completed it successfully. Both endpoints can now continue the connection on the new 4-tuple. A note of importance, the previously used connection IDs (0xaa and 0xbb) must not be reused when sending data via this 4-tuple, both endpoints can send a RETIRE_CONNECTION_ID-frame to their peer to remove them from the connection. See Figure 17 for a visualisation.

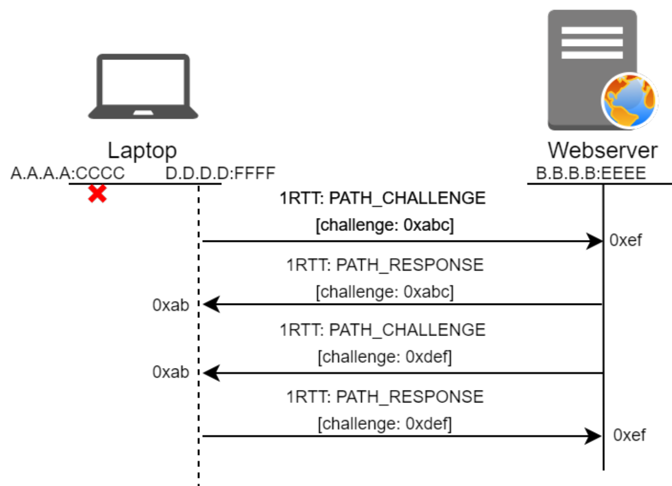


Figure 17: A QUIC connection migrating.

During the procedure it is entirely possible that the path validation fails. This can happen for various reasons, such as packets carrying the frames being dropped. If this happens, the endpoints either have to fall back to the previous 4-tuple, or have to wait for another 4-tuple to become available. If the latter happens, the procedure has to be performed again. Additionally, it is possible for an endpoint to notice that a peer has migrated to a different 4-tuple without changing the connection ID of the packets. This can happen when, for example, a NAT rebinds the IP-address that the peer was using to a different one. In this case, the endpoint should react to this change by switching to a different connection ID to send data to the peer, and perform a path validation procedure. This allows the peer to be informed about a possible, in this case, NAT-rebind, for which the peer might also perform its own path validation procedure.

5 MultiPath QUIC (MPQUIC)

As discussed in section 4, the QUIC transport protocol contains mechanisms that aim to solve the existing problems of TCP. However, not all features for QUIC were added in the current design of the protocol. While thinking of mechanisms that should be implemented in QUIC, the idea of MultiPath was also discussed. It was known that a MultiPath extension had been proposed and deployed for TCP, and that SCTP also already featured it. The concept of MultiPath could be integrated as a mechanism in the current design of QUIC, but the choice was made to stick with a single-path design instead of using multiple paths. The reason for this decision was that the focus of QUIC should be on solving core problems such as, but not limited to, implementing security and encryption for header values, allowing the connection to migrate to different 4-tuples, reducing connection setup time and handling the TCP HoL-blocking problem. Additionally, the use of multiple paths would also result in a big increase in complexity because each path would have different behaviour, making for example the fine-tuning of a congestion controller very difficult.

This did not mean that the idea of MultiPath in QUIC was scrapped entirely, instead it was decided to postpone the addition of MultiPath in QUIC to a later version, such as QUIC v2 [19]. However, there are developers that have interests in implementing MultiPath in QUIC, and they could not wait for a new development cycle of QUIC. Therefore, they have made multiple design proposals to implement the concept. As of writing this thesis, these proposals all have been created in the form of an extension to the current design of QUIC. A first design was proposed by Q. De Coninck and O. Bonaventure [20], a second design was proposed by Q. An, Y. Liu and Y. Ma [21], while a last third design was proposed by C. Huitema [22]. For the remainder of this thesis the first design, proposed by Q. De Coninck and O. Bonaventure, will be the focus. This is due to the other designs being published only recently (after the work on this thesis started). A discussion on the differences between this approach and the other two designs will be provided whenever applicable.

5.1 MultiPath in QUIC

As was the case with regular TCP, a QUIC connection between two endpoints would exist on a single path that is defined by one IP-address from each peer. As discussed before, modern systems often have multiple IP-addresses available, so just like MPTCP, the MPQUIC-extension should also be able to identify unused pairs of IP-addresses and use them to allow peers to communicate and transfer data over multiple paths. While both extensions aim for the same goals due to the same concept of MultiPath being used, their implementations differ between the two protocols. The reason is because the extension for QUIC is able to use the new mechanisms that are provided by the protocol, whereas the extension for TCP had to be compatible with current implementations of TCP and the behaviour of middleboxes.

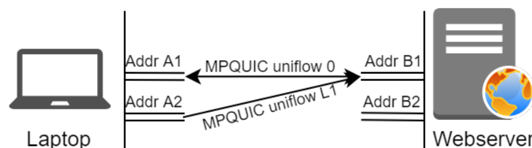


Figure 18: An example MPQUIC connection.

Changing the given scenario to work with QUIC: A modern laptop that can connect to a network over a wired connection, as well as a different local WiFi-network. The laptop would then have two IP-addresses available: A1 and A2, one for the wired connection and one for the WiFi-network. As with MPTCP, the laptop is going to download a large file from some remote webserver over the Internet. This webserver is also accessible via two IP-addresses: B1 and B2. A regular QUIC connection could then exist on, for example, the address combination A1 and B1. A MultiPath QUIC connection would see this as uniflow 0 in both directions. The extension could add additional uniflows, for example, on the combination A2 and B1 (uniflow L1). See Figure 18 for a visualisation.

5.2 Requirements for MultiPath in QUIC

Before the concept of MultiPath can be implemented as an extension for QUIC, some thought must be given to the current design of the QUIC protocol. As stated in the requirements of MP in section 2.3, all MP signalling must be done using an already existing connection between two endpoints. With QUIC these signals can be communicated by defining new frames that use a unique type-value. QUIC already provides a set of standard frames and mechanisms to place frames in (and extract frames from) packets that are transported over the connection with the peer. Like all other frames in QUIC, the size of these newly defined MP-frames should be kept as small in size as possible to reduce the number of bytes that need to be transported.

While MultiPath is still being implemented as an extension for QUIC, it is entirely possible for an endpoint to have no support for MultiPath at all, due to it not being a part of the default implementation of QUIC. Thus, whenever a new connection is initialized, an endpoint should confirm that its peer supports the MultiPath extension. In QUIC this can be done with the use of transport parameters during the cryptographic handshake of a connection, where both endpoints are able to add an additional transport parameter to signal that it supports MultiPath. Additionally, if an endpoint receives the transport parameter that signals MultiPath support and it does not support the extension it can safely ignore the parameter. Finally, if an endpoint receives the expected transport parameter during the cryptographic handshake it can assume that the connection will be supporting additional MultiPath frames.

While the solutions for the previous two requirements have differences between TCP and QUIC, some similarities can be seen when comparing the requirements in general: Both solutions use some form of an extendable mechanism to communicate the MultiPath signals between the two endpoints, and in both cases an endpoint needs to confirm that its peer supports the extension. However, not all requirements that need to be met for TCP should also be considered for QUIC. A big problem for MultiPath TCP was the fact that middleboxes could perform unexpected actions that could break the entire subflow, or even the entire MultiPath connection. Thus, TCP had to implement mechanisms like checksums and a fallback operation to prevent the connection from becoming unusable (see section 3.2. In MultiPath QUIC there is no need to provide such mechanisms to cope with middleboxes, because the default design of QUIC already provides protection against those middlebox operations by encrypting almost everything that is sent over the wire, including header values.

5.3 Initiating a MPQUIC connection

A MultiPath QUIC connection starts in the same way as a regular QUIC connection. It is initiated via a handshake phase during which the client and server establish shared secrets using the cryptographic handshake. Additionally, transport parameters are exchanged, which can be split in two categories. The first category are transport parameters that define restrictions to which an endpoint must adhere, such as the "max_udp_payload_size" parameter that defines the maximum number of bytes a QUIC packet can encompass. The second category are new transport parameters that can be defined to negotiate new protocol behaviour, so in the case of MultiPath a new transport parameter called "max_sending_uniflow_id" is defined to signal support for the MultiPath extension.

From the scenario's perspective: The laptop connects to the remote webserver by sending an initial packet carrying a CRYPTO-frame to the webserver. The CRYPTO-frame contains a cryptographic client_hello and a set of transport parameters and their values. In this set the laptop also adds the "max_sending_uniflow_id"-parameter with a value of 1 to signal MultiPath support. The exact meaning of this value is discussed in section 5.4. The packet is also marked with a source connection ID (0xaa) and a random destination connection ID (0xff) that were generated by the laptop. The webserver receives this packet and creates a context for the new connection. It then stores the laptop's connection ID (0xaa) and generates its own source connection ID (0xbb).

The webserver replies with multiple packets, the first one being an initial packet carrying a CRYPTO-frame containing a cryptographic server_hello, and an ACK-frame. The second packet is a handshake packet carrying a CRYPTO-frame and a set of transport parameters and their values. In this set the webserver has added the "max_sending_uniflow_id"-parameter and a value of 3 to signal that it also supports MultiPath. The third packet is a 1RTT packet carrying STREAM-frames to indicate what type of streams the webserver has opened. All packets are marked with the webserver's source connection ID (0xbb) and the laptop's destination connection ID (0xaa).

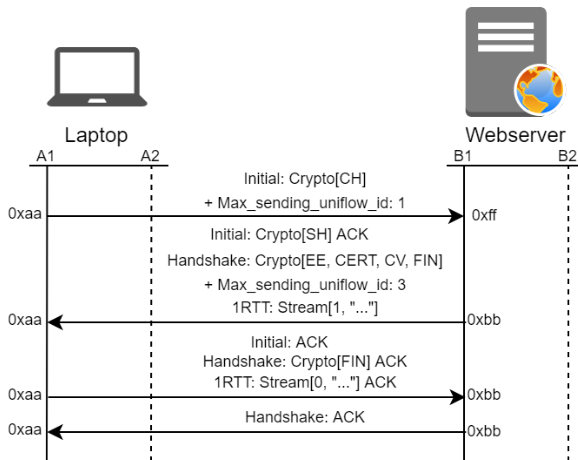


Figure 19: Creating a MPQUIC connection.

After processing the three packets and storing the webserver's connection ID and transport parameters, the laptop responds with three packets as well, the first one being an initial packet carrying an ACK-frame. The second packet is a handshake packet carrying a CRYPTO-frame and an ACK-frame. Lastly, the third packet is a 1RTT packet carrying 1RTT data, such as STREAM-frames and NEW_CONNECTION_ID-fames. All packets are marked with the laptop's source connection ID (0xaa) and the webserver's destination connection ID (0xbb).

In the last step the webserver sends a final handshake packet carrying an ACK-frame. The packet is marked with the webserver's source connection ID (0xbb) and the laptop's destination connection ID (0xaa). From now on, both endpoints have established a connection to transport data to the peer. See Figure 19 for a visualisation.

However, it is possible that the remote webserver does not support the MultiPath extension; in this case the webserver is able to safely ignore the additional transport parameter that the laptop included in its initial packet, and to continue the handshake to set up a connection. The laptop will receive a handshake packet that does not include the additional transport parameter, leading to it disabling the extension so that both endpoints can communicate over a regular QUIC connection. The big difference with TCP is the fact that all transport parameters are integrity protected with some additional checksum, meaning that middleboxes are unable to strip the "max_sending_uniflow_id"-parameter.

5.4 From using bidirectional paths to creating uniflows

Before explaining the meaning of the new transport parameter value for MultiPath, a new concept named "uniflows" needs to be introduced. In section 4 it is mentioned that a regular QUIC connection exists on a single, bidirectional path between two endpoints that is temporarily bound to a 4-tuple. Both endpoints use connection IDs to identify the connection from their perspective, and with the path validation and connection migration mechanisms in QUIC the connection is able to change to a different path (and 4-tuple) during its lifespan. This already highlights the idea that multiple paths can exist between two endpoints.

To use these multiple paths simultaneously, certain mechanisms need to be provided by the MultiPath extension. However, the extension takes a slightly different approach than TCP, where in effect additional TCP connections are set up and signalled. Instead, the extension does not define a regular QUIC connection as a single bidirectional path, but as a composition of two independent, unidirectional packet flows (or two "uniflows"), as seen in Figure 20. The first uniflow (L0) transports packets from the client to the server, while the second uniflow (S0) transports packets from the sever to the client.

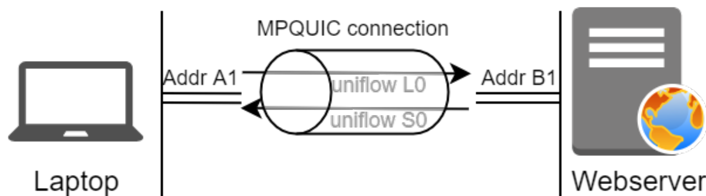


Figure 20: An initial MPQUIC connection with two uniflows.

Thus, whenever a new QUIC connection is created, both endpoints actually create their own initial uniflow to send data to their peer, and both uniflows use the same path that is temporarily coupled to a 4-tuple. An endpoint no longer sends data directly over a path, instead it sends data over a uniflow that is currently using a given path. As visualised in the figure above, both the laptop and the webserver will have different views over these two initial uniflows, where they make a distinction between a sending and a receiving uniflow. The laptop will see L0 as a sending uniflow and S0 as a receiving uniflow, whereas the webserver will see L0 as a receiving uniflow and S0 as a sending uniflow. With this definition of a QUIC connection an endpoint can perform MultiPath by creating and using additional uniflows besides the initial uniflow to send data to their peer, where each uniflow can use a (different) given path that is coupled to a 4-tuple.

To ensure that data is correctly transported over a uniflow, both the sending and receiving endpoint will need to identify a uniflow, keep track of its context (such as to which 4-tuple it is currently coupled), and communicate information about its context with their peer. Unlike TCP, the 4-tuple can change when a QUIC connection performs a migration. In MPQUIC, the sending uniflows are also able to individually perform migrations, thus the 4-tuple cannot be used as an identifier for a specific uniflow. As a first solution, the MultiPath extension solved this problem by allocating an ID to each sending and receiving uniflow, starting from 0 and increasing by 1 up to a chosen maximum value of N. Chosen by the authors of the draft, the ID of a uniflow was added to the header of a packet, which allowed the sender to explicitly tell the receiver which uniflow was used to transport the packet, even when a migration has occurred. Furthermore, to communicate information about the context of a uniflow, the ID of that uniflow is also added to certain newly defined MultiPath frames. The names and mechanisms behind the use of these new frames are explained in the following subsections. The value of N can be chosen by each endpoint as a maximum number of uniflows that can be used by the endpoint to send data to its peer, and also becomes the value of the "max_sending_uniflow_id"-transport parameter that is communicated by each peer during the handshake phase. When an endpoint receives the transport parameter with a given value of N, it knows that the peer wants to use the MultiPath extension and it can conclude that the peer wants to use at most N+1 uniflows to send data. Both endpoints are allowed to communicate different values for the transport parameter; this allows for an asymmetric design where an endpoint can choose any upper bound on the number of sending uniflows it wants to support. With this design it is possible for an endpoint to even communicate a value of 0 for the transport parameter, informing the peer that the endpoint wants to support MultiPath, but that it does not want to use any additional uniflows to send data on this connection.

From the scenario's perspective: The laptop gave the "max_sending_uniflow_id"-parameter a value of 1, signalling to the webserver that it wants to use the MultiPath extension for QUIC and that it wants to use at most 2 uniflows to send data. The webserver gave the transport parameter a value of 3, confirming that it also wants to use the extension and that it wants to use at most 4 uniflows to send data to the laptop. Putting this together, in an optimal connection where every uniflow has been established, the laptop will have 2 uniflows to send data (L0 and L1), and 4 uniflows to receive data (S0 to S3), while the webserver will have 2 uniflows to receive data (L0 and L1), and 4 uniflows to send data (S0 to S3). There is thus a one-to-one mapping between the sending uniflows from one endpoint and the receiving uniflows from the other endpoint. See Figure 21 for an abstract visualisation.

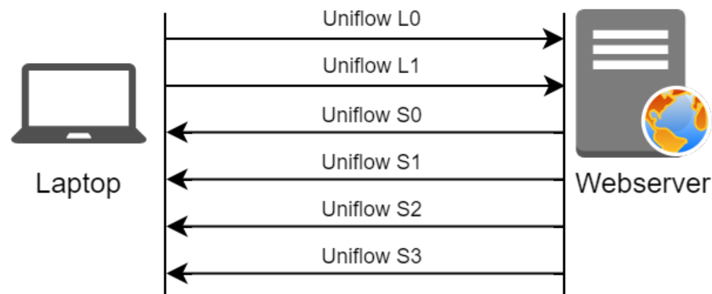


Figure 21: An abstract overview of the possible uniflows.

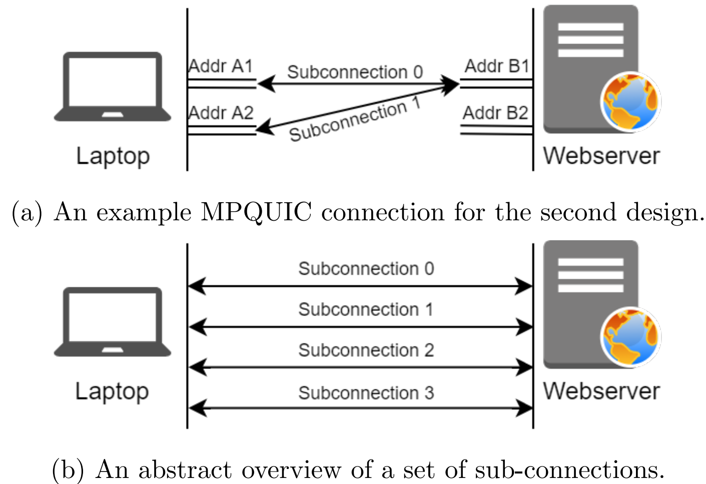
When comparing this idea for a MultiPath design in QUIC with the MultiPath design for TCP, one can see that this design deviates strongly from the already standardized design for TCP. The origin of this deviation can be traced back to the different perspectives that each design has regarding a “path”. Since the early days of TCP, a path was denoted by a 4-tuple (Source IP-Address, Source Port Number, Destination IP-Address, Destination Port Number) that identified a TCP connection. Both endpoints would use that single path to send data to each other, therefore also seeing it as a bidirectional path. The MultiPath TCP design took this perspective and multiplied it, resulting in multiple bidirectional paths where endpoints could communicate with each other.

In QUIC, a path is only seen as a UDP path from the local host to the remote one. Once a QUIC connection has been established, both endpoints have their own view of the 4-tuple used to send packets and the 4-tuple on which it receives packets. The reason for these different views comes from the fact that middleboxes, such as NATs, may alter the 4-tuple of a packet. Due to these different views over the 4-tuple that denotes a path, the authors for this MultiPath design made the argument that a QUIC connection does not use a path to communicate in both directions, and instead uses a composition of two independent, unidirectional packet flows (thus, uniflows), where each flow is coupled to one view of the 4-tuple.

By applying MultiPath to this idea, the authors defined a QUIC connection as a collection of uniflows, where the number of uniflows in one direction does not have to be the same as the number of uniflows in the other direction. For example, one endpoint might not need many uniflows to send data, because it does not have much data to send in the first place. However, this endpoint might have many resources available to receive data, and the other endpoint might have a lot of data that needs to be sent, potentially needing multiple uniflows to be used.

By allowing both endpoints to use a different number of uniflows to send data to each other, the design allows for more fine-grained control over what the protocol is allowed to do. An endpoint is no longer obligated to provide both a sending and receiving context when an additional path is used by its peer. Additionally, the unidirectional nature of these paths now also allows for unidirectional links to be used, such as non-broadcast satellite links.

Further comparing this concept with the other designs mentioned in section 5.1, there is one similarity that can be identified across all of them. The three designs each use a transport parameter (as explained in the requirements) to communicate the support for some form of a MultiPath operation. This is also where the similar perspectives end, because the second design of Q. An et al. builds on the concept of bidirectional sub-connections, instead of uniflows. In this design a connection can contain one or multiple sub-connections, identified by a Sub Connection Identifier (SCI), which are bidirectional and linked to a certain path. The idea behind sub-connections is similar to the currently existing standard for MultiPath TCP, where each sub-connection allows both endpoints to send data to each other. This results in a symmetric design (as seen in Figure 22) when compared to the asymmetric design of uniflows. For this design a new "max_sub_conn_index"-transport parameter is defined, carrying the maximum number of sub-connections that can exist in the current connection. The mechanisms to create, send data, and close additional sub-connections are also different from the design of Q. Deoninck.



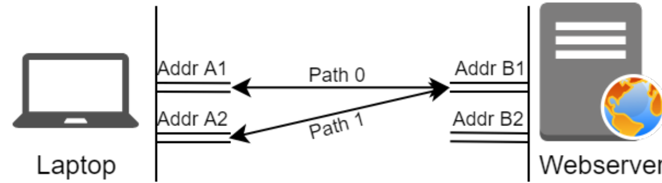
(a) An example MPQUIC connection for the second design.

(b) An abstract overview of a set of sub-connections.

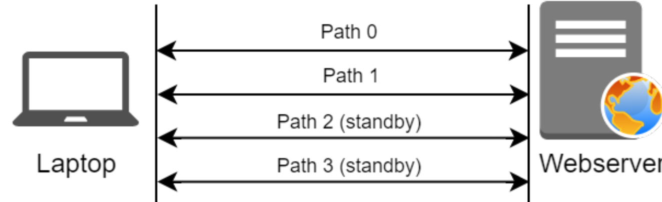
Figure 22: Representation of a MPQUIC connection via design 2.

The approach in the third design by C. Huitema is even more different than the previous two, as it does not introduce additional concepts like uniflows or sub-connections to the current design of QUIC. Instead, it focuses on the idea of introducing an active path management mechanism, replacing the current passive mechanism in QUIC that discards a path and its context when a peer performs a migration. The new active mechanism keeps track of a set of paths, identified by a Path ID (as seen in Figure 23), that are being used by the peer to send data. This design is also symmetrical like the second design, however no comparison to the current standard for MultiPath TCP can be made. For this design, a new "enable_path_management"-transport parameter is defined, carrying a value that signals whether an endpoint allows reception of and/or will send design specific frames during a connection.

All three designs have their own approach on implementing MultiPath in QUIC. Comparisons in the following sections support the idea that the scope and complexity of a design shrinks when moving from the first design to the second one, and even to the third design.



(a) An example MPQUIC connection for the third design.



(b) An abstract overview of a set of paths.

Figure 23: Representation of a MPQUIC connection via design 3.

5.5 Using additional uniflows

The solution to identify subflows via IDs when transporting packets between two endpoints does come with several drawbacks. The first drawback that it brings is the fact that the MultiPath extension alters the semantics of a 1RTT packet in the QUIC protocol by adding an additional field to the packet header. A second drawback is also immediately introduced, because the additional field was added to the public part of the header. These two drawbacks lead to multiple concerns regarding the privacy and non-detectability of a QUIC connection. Firstly, an attacker could identify the individual uniflows that belong to the same connection by checking if each uniflow uses the same connection ID, but different uniflow IDs, resulting in "linkability". Secondly, by exposing the uniflow IDs to the public, intermediate middle-boxes could start parsing the QUIC packets and dropping them if the header field does not have an expected value, resulting in "ossification". More information about linkability and ossification can be found in [23].

To prevent these issues from being exploited, a different approach needs to be utilized to signal the identity of a uniflow when sending a packet. The sender still needs to provide some information when sending data over a uniflow to help the receiver identify on which uniflow it currently is receiving the data. By looking at a regular QUIC connection and applying the extension's idea that it exists out of two initial uniflows, the QUIC protocol already indirectly provides the sender with a different solution for this problem, when an endpoint uses its initial uniflow to send data. This solution stems from the fact that the connection (and indirectly, the initial uniflow) can be identified via the set of connection IDs, which were given by the receiver. The extension can further expand upon the meaning of these connection IDs, by allowing not only the identification of a connection and its initial uniflows, but also the identification of any additional uniflows. Thus, both endpoints provide their peer with uniflow-specific connection IDs, so that both endpoints have a set of connection IDs for each of their sending uniflows.

By using connection IDs that identify a specific uniflow the extension now prevents linkability of different uniflows in a connection. By using different connection IDs for each uniflow an attacker will have a difficult time to identify all the uniflows in a given connection, because no universal connection ID is used by multiple uniflows, and the used connection IDs were communicated beforehand in an encrypted manner via an already existing uniflow. On top of this, the use of uniflow-specific connection IDs does not require any public header changes, preserving the QUIC invariants and preventing ossification with middleboxes.

By applying this solution to the extension, the creation and use of additional uniflows can now be explained. When an initial, MultiPath-supporting connection between two endpoints has been established via the handshake phase, it is impossible for an endpoint to immediately create and use additional uniflows to send data to their peer. This is due to the endpoint not yet having a connection ID that signals the identity of any additional uniflow to the peer. Upon inspecting the current state of the connection, an endpoint only has a single connection ID available for use when sending data over its initial sending uniflow. This single connection ID was given by the peer during the handshake, signalling that the peer will be able to identify both the connection and the initial uniflow upon receiving a packet that is marked with that connection ID. Thus, the extension must introduce a mechanism which allows the exchange of additional connection IDs that signal the identity of the additional uniflows.

This new mechanism was not designed from scratch, because the QUIC protocol already provides a mechanism that can be used by an endpoint to communicate additional connection IDs to their peer. Initially designed to support a peer in migrating to a different 4-tuple, the mechanism provides the ability to create, transport, and process NEW_CONNECTION_ID-frames between two endpoints (See section 4.1 for an explanation). There was one problem however. The mechanism only allowed for connection IDs to be exchanged that identified the initial uniflows of a connection. To preserve the semantics of the already existing NEW_CONNECTION_ID-frame the extension introduced a new frame, called the MP_NEW_CONNECTION_ID-frame. This new frame carries the exact same information as its old counterpart, except for one additional field: the uniflow ID, that carries the ID of a uniflow for which the endpoint will use the connection ID to identify it. Giving the uniflow ID a value of 0 results in the new frame meaning the same as a regular NEW_CONNECTION_ID-frame.

With this new frame in place an endpoint could allow its peer to start using additional uniflows to send data. The only thing the endpoint has to do is generate additional connection IDs for a specific uniflow and send them to its peer via a 1RTT packet carrying MP_NEW_CONNECTION_ID-frames, containing the newly generated connection IDs. However, the extension was designed to allow for fine-grained control while being enabled. The first bit of control came in the form of limiting the sender in only using as many sending uniflows as the receiver was willing to support with receiving uniflows. Thus, an endpoint can choose how many sending uniflows their peer could use by only generating and sending connection IDs that identify a number of receiving uniflows for which they generated a context. An endpoint might decide to allow fewer receiving uniflows to be used than that their peer advertised during the connection's handshake phase. The reason behind this decision can vary, for example from system policy to having insufficient resources available.

While an endpoint is contemplating the creation of additional receiving uniflows, generating connection IDs, and sending packets carrying MP_NEW_CONNECTION_ID-frames to their peer, that peer is also doing the exact same thing. So, the endpoint is also allowed to create a context for each sending uniflow that it wants to use. Upon receiving a packet from its peer that carries MP_NEW_CONNECTION_ID-frames the endpoint is able to mark the identified sending uniflow as unused. The next step that the endpoint needs to complete, before the uniflow can actually be used to send data, is to bind the uniflow to a chosen 4-tuple and verify whether the endpoint can reach its peer via the path that is based on the 4-tuple.

After choosing one of its own IP-addresses to combine with an IP-address from the peer the endpoint is able to start a path validation procedure to ensure connectivity for a new sending uniflow (See section 4.2 for an explanation). The endpoint generates a random challenge value that is placed in a PATH_challenge-frame, which in turn is placed in a 1RTT packet marked with the uniflow's connection ID. The packet is then encrypted and sent over the sending uniflow, from the chosen local IP-address to the chosen remote IP-address. If the peer receives this packet, and the packet is marked with a valid connection ID, the correct receiving uniflow and connection are identified. The peer will then try to decrypt the packet and extract the challenge value. Afterwards, the peer will place that value in a PATH_RESPONSE-frame which in turn is placed in a 1RTT packet. From here, the path validation mechanism deviates from the standard implementation in regular QUIC, because the peer transmits the packet back to the endpoint via an already existing uniflow (for example, the initial uniflow).

Once the endpoint receives this packet, and is able to identify the receiving uniflow and the connection, and can decrypt the packet, the challenge value from the PATH_RESPONSE frame is compared to the challenge value that was generated earlier. If both values are confirmed equal the endpoint can finish the path validation procedure and mark the sending uniflow as active. From this point onward the endpoint can select the uniflow to send data to its peer.

Combining all this together, from the scenario's perspective: The laptop and the webserver have performed a handshake and established a MultiPath-supporting QUIC connection. At this point, the laptop received a connection ID with a value of 0xbb that identifies the initial receiving uniflow on the webserver. The laptop also gave the webserver a connection ID with a value of 0xaa, identifying the initial receiving uniflow on the laptop. Both initial uniflows are currently bound to the IP-address combination A1 and B1. The webserver identifies that the laptop wants to use an additional uniflow to send data, based on the received "'max_sending_uniflow_id"-parameter value.

The webserver decides that it wants to allow the laptop to use two uniflows to send data. Therefore, it generates a new context for the additional receiving uniflow and allocates an ID value of 1 to the uniflow. The webserver then generates a new connection ID with a value of 0xcc and places the connection ID in a MP_NEW_CONNECTION_ID-frame. The frame is also given a value of 1 for the uniflow_id-field, this tells the laptop that the new connection ID can be used when sending data over uniflow 1. The webserver places the frame in a 1RTT packet and encrypts the packet, marking it with the laptop's destination connection ID (0xaa). Finally, the webserver sends the packet to the laptop over sending uniflow 0.

The laptop receives the packet and is able to identify that its initial receiving uniflow received it. The laptop then decrypts the packet and reads the MP_NEW_CONNECTION_ID-frame. The connection ID is then added to the context of its own sending uniflow with ID 1. At this point the laptop marks the uniflow as unused, indicating that a path validation procedure can be performed for that uniflow. The laptop sends back a 1RTT packet to the webserver over sending uniflow 0, carrying an ACK-frame and marked with the destination connection ID of the server (0xbb). After this, the laptop identifies that it has an additional IP-address available, A2. It also identifies that the address combination of A2 and B1 does not exist yet, so it decides to try the path validation procedure for this address combination on sending uniflow 1. The laptop generates a random challenge value (0xabc) and places that value in a PATH_CHALLENGE-frame. The laptop places the frame in a 1RTT packet and encrypts the packet, marking it with the webserver's destination connection ID (0xcc), indicating that the webserver should identify uniflow 1 as the receiving uniflow. Finally, the laptop sends the packet to the webserver over sending uniflow 1.

The webserver receives two packets and identifies that the first packet belongs to receiving uniflow 0, whereas the second packet belongs to receiving uniflow 1. The webserver also sees that the receiving uniflow with ID 1 has not yet been coupled to a 4-tuple, thus it collects the laptop's IP-address and binds the uniflow to the new 4-tuple (A2 and B1). The webserver then decrypts both packets and determines that the previously sent packet, containing the connection ID for uniflow 1, has been successfully received by the laptop. The challenge value is extracted from the PATH_CHALLENGE-frame in the second packet, and placed in a new PATH_RESPONSE-frame. The webserver places the frame in a 1RTT packet and encrypts the packet, marking it with the laptop's destination connection ID (0xaa). Finally, the webserver sends the packet to the laptop over sending uniflow 0.

The laptop receives the packet and is able to identify that its initial uniflow received it. The laptop then decrypts the packet and checks if the returned challenge value of the PATH_RESPONSE-frame equals the previously generated challenge value. If this is the case the laptop marks its sending uniflow with ID 1 as active, confirming that the laptop can reach the webserver over the address combination of A2 and B1, meaning that the uniflow can now be used to send data to the webserver. See Figure 24 for a visualization.

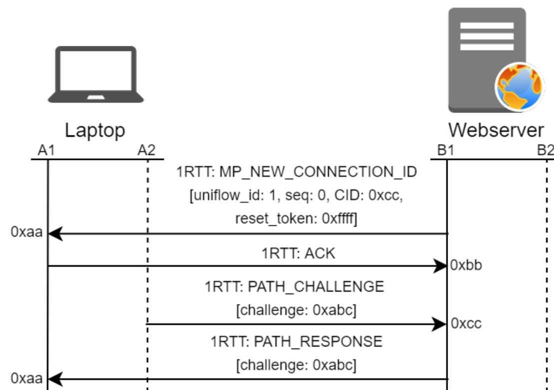


Figure 24: An additional uniflow establishment.

From an external perspective the additional uniflow that is being used to send data looks like another regular QUIC connection between two endpoints. The only observable difference is that no initial or handshake packets have been exchanged over the path. Compared to TCP, these additional paths are also guaranteed to support the transmission of MultiPath frames, because in 1RTT packets everything, besides the public header, is encrypted with 1RTT keys that were generated from an established secret during the handshake phase on the initial uniflows.

Lastly, the new mechanism that introduced uniflow-specific connection IDs also provides the functionality for an endpoint to signal to its peer that it no longer wishes to utilize a particular connection ID that is bound to a certain sending uniflow. Just like an endpoint would send a RETIRE_CONNECTION_ID-frame to retire a given connection ID for sending uniflow 0, the endpoint is also able to retire connection IDs from any additional uniflows by sending a MP_RETIRE_CONNECTION_ID-frame to its peer. The new frame carries information identical to its old counterpart, except for one additional field. This additional field is called the uniflow ID and carries the ID value of a uniflow for which the endpoint wishes to retire a connection ID.

When comparing this design with the other designs mentioned in section 5.1 a couple of interesting similarities and differences can be identified while looking at the creation and use of additional uniflows/sub-connections/paths. For example, the second design by Q. An et al. also allocates some form of identity (the Sub Connection Identifier) to each of the sub-connections that can exist in a QUIC connection. Additionally, the second design also uses connection IDs to identify sub-connections when sending packets. However, the mechanism around the use of connection IDs is different from the design of Q. De Coninck, where uniflow-specific connection IDs were exchanged. Instead, the design only uses a single pool of connection IDs.

After performing the handshake, both endpoints provide their peer with a pool of connection IDs, just like a regular QUIC connection. From here, the client has the ability to create additional sub-connections. The client first searches for unused pairs of IP-addresses. If a new 4-tuple is found, the client moves to the next step, where path validation is performed to ensure that the client can reach its peer via that new 4-tuple. If the path validation procedure succeeds the client can initiate a new sub-connection. These two steps are conceptually the same as in the first design.

However, this is where the second design deviates from the first. In this design, the client initiates a new sub-connection by choosing a destination connection ID from the pool of connection IDs given by its peer, effectively telling the peer that it needs to use that connection ID to identify the new sub-connection. The client then sends a 1RTT packet to the peer (via the new 4-tuple and marked with the chosen connection ID) that contains a MP_NEW_SUB_CONN-frame to communicate the SCI of the sub-connection. The peer then chooses its own destination connection ID from the pool of connection IDs given by the client and sends back a 1RTT packet (via the newly observed 4-tuple and marked with its own chosen connection ID) carrying a MP_SUB_CONN_ACCEPT-frame to signal that it now has the identity of the additional sub-connection.

The peer also must perform a path validation procedure on the observed 4-tuple to ensure that it can reach the client before sending actual data over the new sub-connection. If this path validation procedure succeeds, the new sub-connection is established and ready for use to transport data. See Figure 25 for a visualisation.

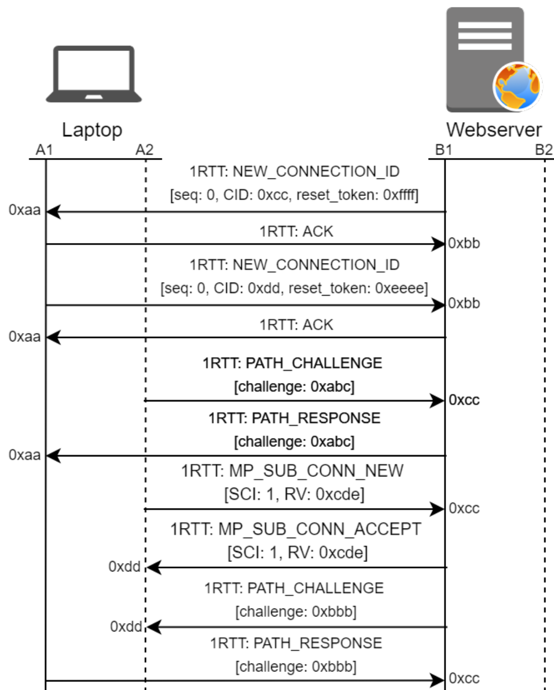


Figure 25: An additional sub-connection establishment.

In this second design, the client can decide at any point during the connection to open an additional sub-connection if an unused 4-tuple is identified, but it does this without knowing whether its peer really wants to allow it. The peer can deny the creation attempt by not sending a MP_SUB_CONN_ACCEPT-frame back to the client, but this leads to a situation where the client can continuously force the peer to decide whether it should create an additional context to support the new sub-connection. Comparing this to the design of Q. De Coninck, it is the peer that first has to give the client permission to create an additional uniflow by sending uniflow-specific connection IDs before the client can attempt to do so, giving more control to both endpoints.

As previously mentioned in section 5.4 the third design by C. Huitema only introduces an active path management mechanism to the protocol. Just like the previous two designs it also identifies each path by an identifier: the path ID. However, the mechanisms to create and use new paths are different and more basic. Firstly, in a similar manner as the previous two designs, if a client identifies that a new 4-tuple can be created from the known IP-addresses of both endpoints, the client can perform a path validation procedure to check if it can reach its peer via that 4-tuple. If the peer responds and the path validation procedure succeeds, the client allocates a path ID to the 4-tuple and sends a 1RTT packet to its peer via that 4-tuple, carrying a PATH_IDENTIFICATION-frame containing that path ID to allow the peer to identify that path with the same path ID.

The peer also performs a path validation procedure to ensure that it can reach the client through that path as well. Once both the client and its peer have established that the additional path exists, some rules around the use of the path can be negotiated. This is done by sending the peer a 1RTT packet carrying a PATH_STATUS-frame, containing the identity of a path, a value indicating the status and a sequence number. See Figure 26 for a visualisation. For the status of a path the client can choose between three options: available, standby and abandon. The first option denotes that the peer can use this path to send data whenever it wants to, the second informs the peer that it should only send data to its peer via that path if it has no other paths available. Lastly, the third option informs the peer that the client will no longer accept data that is being sent over that path, and that the context and resources for that paths should be released from the management mechanism.

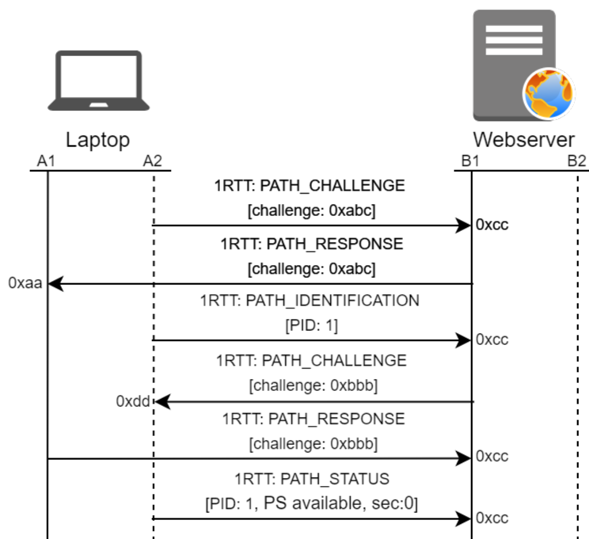


Figure 26: An additional path establishment.

With this path management mechanism in place, once all paths have been identified and their statuses signaled, the design only has to rely on scheduling algorithms to transmit data on specific paths. The client could decide to change the status of a certain path at any time, by sending another 1RTT packet to its peer, carrying a PATH_STATUS-frame containing that path's path ID and a different status value.

5.6 Adding additional IP-addresses

As stated before, an additional sending uniflow can only be initiated and used by an endpoint when its peer has a matching receiving uniflow for which it sent uniflow-specific connection IDs to the endpoint. However, this was not the only prerequisite that needed to be fulfilled. An endpoint also has to find a 4-tuple to temporarily bind the new sending uniflow to. In the current draft there is no restriction that prevents additional uniflows from being bound to the same 4-tuple (such as the 4-tuple of the initial uniflow), but in order to achieve optimal use of the MultiPath concept it is a good practice to bind each sending uniflow to a different 4-tuple.

To find these different 4-tuples, an endpoint needs to know the other IP-addresses of its peer. Therefore, both endpoints need to be able to communicate their additional IP-addresses with each other. The requirements of MultiPath in section 2.3 state that the exchange of IP-addresses can be done in two ways. The first method, being the implicit method, where an endpoint sends a 1RTT packet over a sending uniflow to its peer via an IP-address that the peer does not already know. An example of this method can be seen in Figure 24, where the laptop sends a PATH_CHALLENGE via sending uniflow 1 to the webserver on address combination A2 and B1. The webserver identifies that the package carrying the frame contains a valid connection ID (0xbb) and that it identifies receiving uniflow 1, but that the IP-address A2 has not been seen before. Due to the webserver being able to decrypt the package correctly, it can assume that the packet was sent from the laptop, and that IP-address A2 is one of the additional IP-addresses of the laptop.

The second method to communicate IP-addresses is more explicit, where an endpoint sends a 1RTT packet containing an ADD_ADDRESS-frame to its peer via an already established sending uniflow (such as the initial sending uniflow). This new frame contains an IP-address, along with additional information such as the interface type, the IP-version and a sequence number. By providing this ADD_ADDRESS-frame both endpoints are able to communicate their additional IP-addresses to each other via the two methods. However, this second method also introduces a new problem, because an endpoint could have advertised a private IP-address that resides behind a NAT. Initially, this is already solved by the already existing mechanisms of QUIC, because the path validation procedure that is performed by the peer will always fail for that private IP-address. This is due to the peer not receiving an answer on its 1RTT packet carrying a PATH_CHALLENGE-frame, and therefore declaring that the path validation has failed due to a timeout. Even if there is a third endpoint on a network that can be reached by the peer on that IP-address, that third endpoint will not be able to respond with a PATH_RESPONSE-frame, because it cannot decrypt the 1RTT packet in which the PATH_CHALLENGE-frame was sent.

While this solution already covers the problem of communicating private IP-addresses, the design of Q. De Coninck provides an additional mechanism to use these advertised private IP-addresses, by changing their values with that of their public counterpart. This is done by introducing two new parts, the first one being address IDs that are used to identify a specific IP-address, even when the value of the IP-address has changed. Additionally, address IDs will allow endpoints to communicate about a specific IP-address without having to check whether the IP-address value has been changed. To allow an endpoint to inform its peer about which address is identified by which address ID, the address ID is added to the ADD_ADDRESS-frame that was explained previously.

Thus, from the scenario's perspective: Right after performing the connection's cryptographic handshake, and before establishing its second sending uniflow, the laptop decides to communicate its second IP-address (A2) to the webserver. To do this, the laptop allocates the value of 1 as A2's address ID. The laptop then places the IP-address, along with the address ID and some additional information, in a ADD_ADDRESS-frame. Afterwards, the laptop places the frame in a 1RTT packet and encrypts the packet, marking it with the webserver's destination connection id (0xbb). Finally, the laptop sends the packet to the webserver over sending uniflow 0.

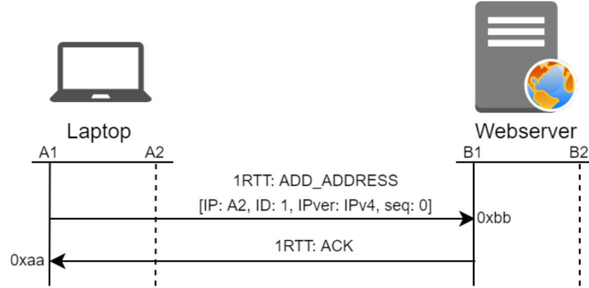


Figure 27: Sending a ADD_ADDRESS frame.

The webserver receives the packet and is able to identify that its initial receiving uniflow received it. The webserver then decrypts the packet and reads the contents of the ADD_ADDRESS-frame. The IP-address and its address ID are stored, and the webserver sends back a 1RTT packet to the laptop over sending uniflow 0, carrying an ACK-frame and marked with the destination connection ID of the laptop (0xaa). See Figure 27 for a visualisation. At this point, the laptop does not know that its WiFi connection, for which it uses IP-address A2, actually resides behind a NAT. This results in the laptop communicating the private representation of A2. After sending the ADD_ADDRESS-frame to the webserver, the laptop also performs the establishment of its second sending uniflow, resulting in the webserver having collected two additional IP-addresses from the laptop besides the initial IP-address A1. See Table 1 for an overview of these IP-addresses.

	IP-address	address ID	linked uniflows
Perceived remote addresses	A2(public)	/	Receiving uniflow 1
Received remote addresses	A1	0	Receiving uniflow 0 Sending uniflow 0
	A2(private)	1	None

Table 1: The webserver's perspective on remote addresses after ADD_ADDRESS.

This table is split up in two categories, being the "perceived remote addresses" and the "received remote addresses". The first category contains a single entry, where the public representation of IP-address A2 is stored. This IP-address is not identified by an address ID, but the webserver has identified that the laptop uses this IP-address to send data over its sending uniflow 1, leading to the webserver receiving that data on the matching receiving uniflow 1. The second category contains two entries, the first entry containing the representation of IP-address A1, which is identified by address ID 0. (Upon performing the cryptographic handshake both endpoints identify the remote IP-address that is used by the peer as address ID 0.) The webserver has identified that this IP-address was used to perform the cryptographic handshake, and that both the initial uniflows still use this IP-address to transport data. Therefore the sending and receiving uniflow 0 are linked to this IP-address. The second entry contains the private representation of IP-address A2, which is identified by address ID 1. There are currently no uniflows linked to this IP-address.

The next step would be to replace the value of the IP-address in the second entry of "received remote addresses" with that of the IP-address in the single entry of "perceived remote addresses". This is where the second part for the mechanism is introduced: an endpoint is allowed to send UNIFLOWS-frames to its peer upon establishing a sending uniflow, informing the peer which active sending and receiving uniflows are using which local IP-addresses on the endpoint. This UNIFLOWS-frame contains two sections, one for the endpoint's active sending uniflows, and one for the endpoint's receiving uniflows. For each section, the endpoint inserts the ID of the sending/receiving uniflow, along with the address ID of the uniflow's local IP-address. For active sending uniflows that send packets, this is the source IP-address of a packet, whereas for receiving uniflows that receive packets, this is the destination IP-address of a packet.

From the scenario's perspective: After establishing its second sending uniflow with ID 1, the laptop decides to inform the webserver about the current IP-address that this uniflow uses to send packets, which is A2. Therefore, the laptop generates a UNIFLOWS-frame and places sequence number 0 inside it. After this, the laptop identifies that it has one receiving uniflow and two sending uniflows active. Thus, the laptop places the values 1 and 2 in the UNIFLOWS-frame, indicating that the frame will contain one entry for a receiving uniflow, and two entries for sending uniflows.

For the receiving uniflow's entry the laptop places the values of 0 and 0 in the frame, indicating that its receiving uniflow 0 uses the IP-address that is marked with ID 0 (thus, A1) to receive packets. For the first entry of the sending uniflows the laptop also places the values of 0 and 0 in the frame, indicating that its sending uniflow 0 uses the IP-address with ID 0 (thus, A1) to send packets. Notice that these are the same values as in the entry of the receiving uniflow, but these have a slightly different meaning. For the second entry of the sending uniflows the laptop places the values of 1 and 1 in the frame, indicating that its sending uniflow 1 uses the IP-address with ID 1 (thus, A2) to send packets. The laptop then places the frame in a 1RTT packet and encrypts the packet, marking it with the webserver's destination connection ID (0xcc). Finally, the laptop sends the packet to the webserver over sending uniflow 1. (This packet could have been sent over either sending uniflow, provided that the correct connection ID is used to mark the packet.) See Figure 28 for a visualisation.

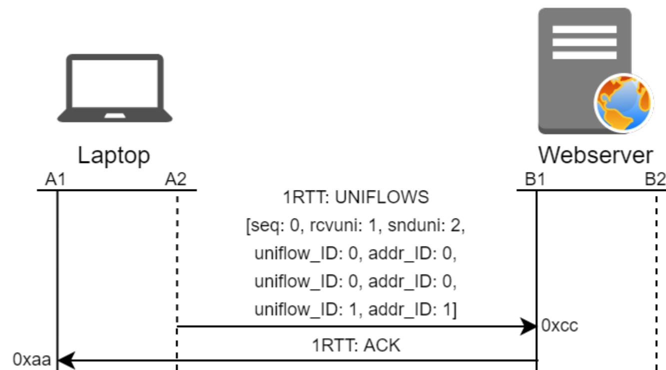


Figure 28: Sending a UNIFLOWS frame.

Upon reception of the packet the webserver identifies that the packet was received on receiving uniflow 1. After decrypting the packet, the webserver reads the contents of the UNIFLOWS-frame and identifies that the IP-address identified by address ID 1 is used by the laptop to send data from sending uniflow 1. The webserver then checks if it has an IP-address that is identified by address ID 1 and finds the private representation of A2. After this the webserver checks if the address ID matches the one that is stored in the context of the receiving uniflow with uniflow ID 1. At this point, the webserver notices that both the address IDs and the IP-address values do not match, and that the address ID that is stored in the uniflow's context is "None".

Based on this information, the webserver is able to assume that the IP-address it received from the ADD_ADDRESS-frame is a private IP-address (with address ID 1), and that the public representation of that IP-address is the one that is currently stored in the context of receiving uniflow 1. Therefore, the webserver takes the value of the IP-address in the receiving uniflow's context and replaces the value of the IP-address received from the ADD_ADDRESS-frame, resulting in a new overview of the IP-addresses as seen in Table 2. Note that the "perceived remote addresses" category is removed because it has no entries.

	IP-address	address ID	linked uniflows
Received remote addresses	A1	0	Receiving uniflow 0 Sending uniflow 0
	A2	1	Receiving uniflow 1

Table 2: The webserver's perspective on remote addresses after linking IP-address A2.

At this point the webserver only stores two entries in the "received remote addresses" category, and no longer keeps track of the entry that was stored in the "perceived remote addresses" category. Note that the linked uniflows from the removed entry are also added to the second entry of the "received remote addresses" category, completing the replacement of the private representation of IP-address A2. As a final step, the webserver also sends back a 1RTT packet to the laptop over sending uniflow 0, carrying an ACK-frame and marked with the destination connection ID of the laptop (0xaa).

With this mechanism that the design of Q. De Coninck has introduced, both endpoints are able to learn each other's public IP-addresses even when private IP-addresses were communicated, leading to a maximized chance of finding 4-tuples that an endpoint can use to reach its peer and set up additional sending uniflows. Comparing this design to the design of MPTCP shows that this is a more complex form of exchanging IP-addressed between two endpoints over a MultiPath connection, due to MultiPath TCP not providing an additional mechanism to replace private IP-addresses with their public counterparts. Unfortunately, at the time of writing this thesis no comparison to the second design of Q. An et al., mentioned in section 5.1 can be made, as it does not provide an explicit mechanism to exchange additional IP-addresses between endpoints. There are mentions of MP_ADD_ADDRESS-frames and MP_REMOVE_ADDRESS-frames, but no explanation or frame format is given. Based on this, while also looking at the design of Q. De Coninck and the design of MPTCP, it can be assumed that the two frames will have the same format as the conceptually equal frames in the first design.

However, it is not known if the second design will also provide an additional mechanism to convert private IP-addresses to their public counterparts. It is likely that this design will choose for a minimal impact on the design of QUIC, where private IP-addresses can still be exchanged but will not be altered, resulting in path validation procedures that will always fail upon using these private IP-addresses (as mentioned previously in this section).

Even the active path manager introduced in the third design of C. Huitema cannot be compared directly to the design of Q. De Coninck. The active path manager will be able to identify and store new IP-addresses when new paths are identified, allowing endpoints to implicitly exchange IP-addresses. However, the mechanism does not provide an explicit method to communicate additional IP-addresses between peers. Therefore, the third design gains the advantage that it does not have to handle private IP-addresses that could be communicated in new frames.

By having different approaches when it comes to communicating additional IP-address the three designs will also have different solutions when it comes to removing these IP-addresses. This will be explained in the next section.

5.7 Removing an IP-address

During the lifetime of a MultiPath QUIC connection it is possible for an endpoint to lose its connection to a network. For example, a mobile phone could be connected to both a WiFi-network as well as 4G. If the mobile phone is moved outside the range of the WiFi-network, it will lose its corresponding IP-address. For QUIC connections in general this does not mean that the connection must be terminated, as was the case with regular TCP. Instead, a regular QUIC connection could, upon losing the WiFi-connection, allow the phone to migrate to the IP-address of the 4G connection as explained in section 4.2. For a MultiPath QUIC connection however, some additional steps need to be taken to ensure that data is no longer being sent by the peer to the lost IP-address.

For the first step, when an endpoint detects that it has lost an IP-address, it must check the 4-tuple of each sending uniflow that is currently marked as active. If the 4-tuple of a sending uniflow contains the IP-address that has been lost, the uniflow is marked as unused and the 4-tuple is removed. After this, the endpoint resets the context of these affected sending uniflows to allow reuse in later stages of the connection. (The relevance of resetting the context is explained in section 5.8.)

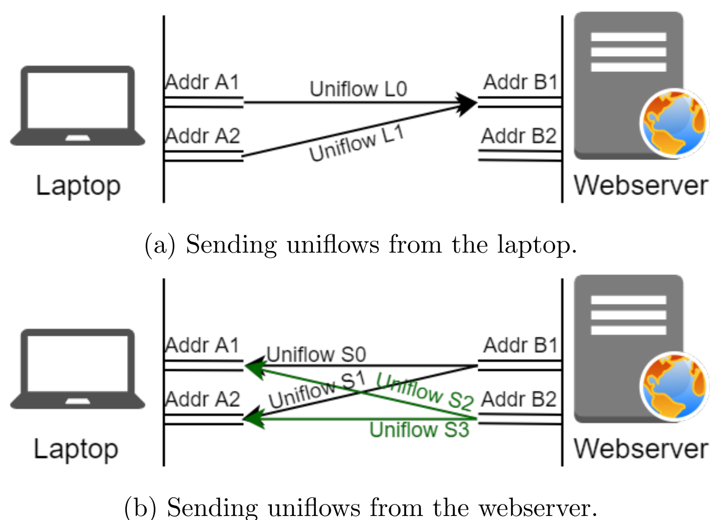
In the second step the endpoint communicates the loss of the IP-address by sending a 1RTT-packet carrying a REMOVE_ADDRESS-frame to its peer. This new frame contains a sequence number and an address ID. The sequence number contains an upwards counting value that is shared with the ADD_ADDRESS-frame to ensure correct handling of these frames when multiple ADD- and REMOVE_ADDRESS-frames are being sent that contain the same address ID. As previously mentioned, the address ID identifies the correct IP-address on the receiving endpoint, regardless of the IP-address's value. Lastly, in the third step the endpoint also sends a new UNIFLOWS-frame to its peer to indicate the status of its remaining active sending uniflows.

Upon reception of the REMOVE_ADDRESS-frame the peer also performs a number of steps. In the first step it must also check the 4-tuple of each sending uniflows. If the 4-tuple contains a remote IP-address with the mentioned address ID the peer marks these sending uniflows as unused, removes the 4-tuple and resets the uniflow's context.

In the second step, the received UNIFLOWS-frame helps the peer to identify which receiving uniflows should be inspected and potentially should be unbound from their 4-tuple, by checking which receiving uniflows are bound to a 4-tuple, but whose uniflow ID is not listed in the sending uniflows info section of the UNIFLOWS-frame. Lastly, the peer also sends back a UNIFLOWS-frame to indicate the status of its remaining sending uniflows. Finally, upon receiving the UNIFLOWS-frame from its peer the endpoint that lost the IP-address can now also check which receiving uniflow contexts should be inspected, and potentially unbind these receiving uniflows from their 4-tuple.

By performing these steps both endpoints are able to identify which sending and receiving uniflows are affected by the loss of the IP-address, and correctly unbind them from their 4-tuples. This prevents the peer from sending data to the lost IP-address, as well as keep a clean context in its receiving uniflows whose 4-tuple was linked to it. Additionally, by allowing the endpoint that lost its IP-address to "reset" the affected sending uniflows by changing their context to unused, additional steps can be performed that are explained after the following example.

From the scenario's perspective: After downloading the large file for a while, both the laptop and the webserver have communicated their additional IP-addresses to each other, as well as established all remaining possible sending and receiving uniflows. An overview of this connection with all the uniflows can be seen in Figure 29 (the green colour has no purpose other than visibility), whereas an overview of the remote IP-addresses in both perspectives can be seen in Tables 3 and 4.



(a) Sending uniflows from the laptop.

(b) Sending uniflows from the webserver.

Figure 29: An overview of all sending (and receiving) uniflows.

As previously mentioned, the laptop's second IP-address (A2) is used to connect to a local WiFi-network. At some point after establishing all the sending uniflows on both endpoints, the laptop is moved to a different position that breaks the connection with the WiFi's access point, which results in the laptop losing IP-address A2. Upon detecting this loss, the laptop proceeds with marking the context of its sending uniflow 1 as unused, and removes the bound 4-tuple. After this, the laptop generates a REMOVE_ADDRESS-frame and places a value of 1 for the sequence number, as well as a value of 1 for the IP-address that is removed.

	IP-address	address ID	linked uniflows
Received remote addresses	B1	0	Sending uniflow 0 Sending uniflow 1 Receiving uniflow 0 Receiving uniflow 1
	B2	1	Receiving uniflow 2 Receiving uniflow 3

Table 3: The laptop's perspective on remote addresses.

	IP-address	address ID	linked uniflows
Received remote addresses	A1	0	Receiving uniflow 0 Sending uniflow 0 Sending uniflow 2
	A2	1	Receiving uniflow 1 Sending uniflow 1 Sending uniflow 3

Table 4: The webserver's perspective on remote addresses.

Note that the sequence number is 1 in this frame because the previous ADD_ADDRESS-frame that the laptop sent for this address had a sequence number value of 0. The laptop then places the REMOVE_ADDRESS-frame in a 1RTT packet and encrypts it, marking it with the webserver's destination connection ID (0xbb). Finally, the laptop sends the packet to the webserver over sending uniflow 0. After sending the packet the laptop also generates a UNIFLOWS-frame and places sequence number 1 inside it. The laptop then places the values 2 and 1 in the frame, indicating that the frame will contain two entries for the receiving uniflows, and one entry for a sending uniflow. For the first entry of the receiving uniflows the laptop places the values of 0 and 0 inside the frame, indicating that its receiving uniflow 0 uses the IP-address with ID 0 (thus, A1) to receive packets. For the second entry of the receiving uniflows the laptop places the values of 2 and 0 inside the frame, indicating that its receiving uniflow 2 also uses the IP-address with ID 0 (thus, A1) to receive packets.

For the sending uniflow's entry the laptop places the value of 0 and 0 in the frame, indicating that its receiving uniflow 0 uses the IP-address that is marked with ID 0 (thus, A1) to send packets. The laptop then places the frame in a 1RTT packet and encrypts the packet, marking it with the webserver's destination connection ID (0xbb). Finally, the laptop sends the packet to the webserver over sending uniflow 0.

The webserver receives these two packets and is able to identify that its initial receiving uniflow received them. After decrypting the first packet, the webserver reads the content of the REMOVE_ADDRESS-frame and identifies that IP-address A2, with address ID 1 has been lost by the laptop. The webserver then inspects each of its sending uniflows and identifies that sending uniflows 1 and 3 have the same destination address ID as the one mentioned in the frame. Therefore, the webserver proceeds with marking the context of these two uniflows as unused, removes the bound 4-tuple from the uniflows and resets their context. After this the webserver decrypts the second packet and reads the content of the UNIFLOWS-frame. Upon inspecting the frame, the webserver sees that the laptop only has one sending uniflow active, which is the initial sending uniflow. With this information the webserver proceeds to unbind its receiving uniflow with uniflow ID 1 from its 4-tuple. After performing this step, the webserver responds back to the laptop by generating its own UNIFLOWS-frame and places sequence number 0 inside it. After this the webserver places values 1 and 2 in the frame, indicating that the frame will contain one entry for a receiving uniflow, and two entries for the sending uniflows.

For the receiving uniflow's entry the webserver places the values of 0 and 0 in the frame, indicating that its receiving uniflow 0 uses the IP-address that is marked with ID 0 (thus, B1) to receive packets. For the first entry of the sending uniflows the webserver also places the values of 0 and 0 in the frame, indicating that its sending uniflow 0 uses the IP-address with ID 0 (thus, B1) to send packets. For the second entry of the sending uniflows the webserver places the values of 2 and 1 in the frame, indicating that its sending uniflow 2 uses the IP-address with ID 1 (thus, B2) to send packets. The webserver then places the frame in a 1RTT packet, along with an additional ACK-frame to acknowledge the previously received packets. The webserver encrypts the packet, marking it with the laptop's destination connection ID (0xaa). Finally, the webserver sends the packet to the laptop over sending uniflow 1. (This packet could have been sent over either sending uniflow, provided that the correct connection ID is used to mark the packet.)

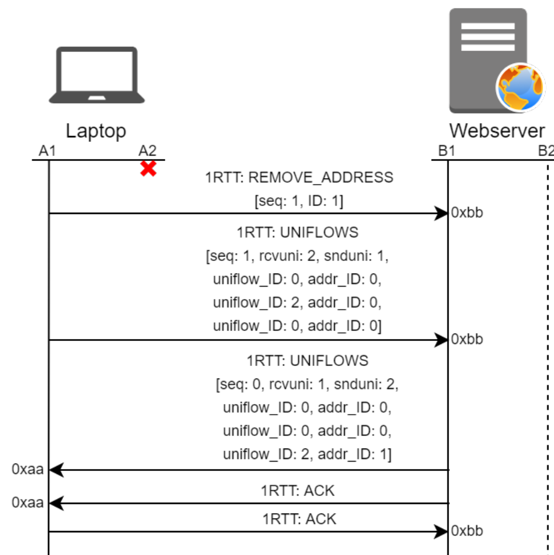
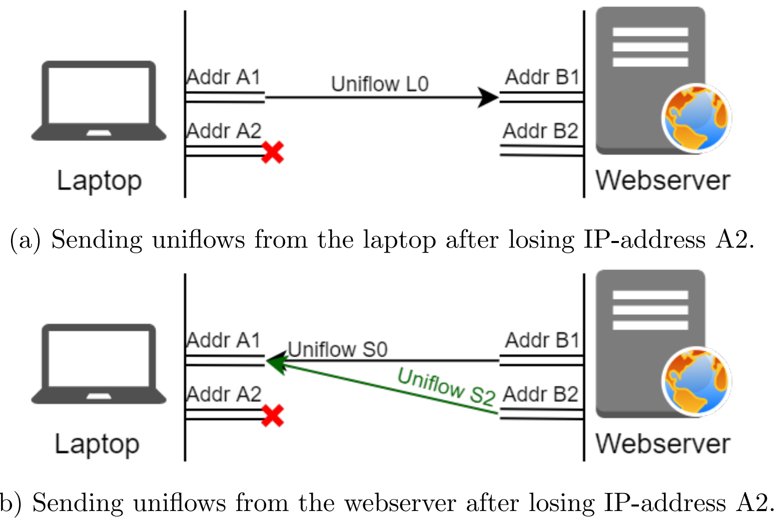


Figure 30: Removing IP-address A2.

The laptop receives the packet and is able to identify that its initial receiving uniflow received them. After decrypting the packet, the laptop reads the content of the UNIFLOWS-frame. Upon inspecting the frame, the laptop sees that two sending uniflows are active on the webserver. Based on this information the laptop is able to identify receiving uniflows 2 and 3 and unbinds them from their 4-tuple, because they have no matching entry in the frame. As a final step, the laptop sends back a 1RTT packet to the webserver over sending uniflow 0, carrying an ACK-frame and marked with the destination connection ID of the server (0xbb). See Figure 30 for a visualisation, see Figure 31 for an overview of this connection and see Tables 5 and 6 for an overview of the remote IP-addresses in both perspectives.



(a) Sending uniflows from the laptop after losing IP-address A2.

(b) Sending uniflows from the webserver after losing IP-address A2.

Figure 31: An overview of all sending (and receiving) uniflows after losing IP-address A2.

<table>
<tr><th></th><th>IP-address</th><th>address ID</th><th>linked uniflows</th></tr>
<tr><td rowspan="2">Received remote addresses</td><td>B1</td><td>0</td><td>Sending uniflow 0
Receiving uniflow 0</td></tr>
<tr><td>B2</td><td>1</td><td>Receiving uniflow 2</td></tr>
</table>

Table 5: The laptop's perspective on remote addresses after losing IP-address A2.

<table>
<tr><th></th><th>IP-address</th><th>address ID</th><th>linked uniflows</th></tr>
<tr><td rowspan="2">Received remote addresses</td><td>A1</td><td>0</td><td>Receiving uniflow 0
Sending uniflow 0
Sending uniflow 2</td></tr>
<tr><td>A2</td><td>1</td><td>None</td></tr>
</table>

Table 6: The webserver's perspective on remote addresses after losing IP-address A2.

By forcing each sending uniflow to be linked to a 4-tuple where each IP-address is identifiable by an address ID, the solution to removing an IP-address becomes easy enough where an endpoint only has to sent two frames to ensure that its peer unlinks all its sending and receiving uniflows from that specific IP-address. In the example it is said that an endpoint utilizes the info of the UNIFLOWS-frame to unlink affected receiving uniflows. However, this can also be done by only using the REMOVE_ADDRESS-frame's info, because the affected uniflows are all linked to the same IP-address with the mentioned address ID. The UNIFLOWS-frame adds the extra pieces of information to ensure that the remaining active matching uniflows use the same IP-addresses.

Comparing this design to the other designs mentioned in section 5.1, some interesting differences can be identified. For example, a mechanism is introduced in the second design of Q. An et al., where it is stated that a sub-connection can be closed upon detecting that one of its IP-addresses has been lost. As said in the previous section, there is currently no mechanism provided for an endpoint to notify its peer about an IP-address that has been lost. Assuming that there will be one in future versions of the design, that is conceptually equal to the mechanism in the design of Q. De Coninck, an endpoint would be able to send a MP_SUB_CONN_CLOSE-frame to its peer to close the sub-connection that is identified by a given Sub Connection Identifier. The design of Q. De Coninck also has a potential frame that can be used, even though it is not one of its intended uses. A sending uniflow can be "closed" by the endpoint by unbinding the 4-tuple and performing a context reset. Afterwards a UNIFLOWS-frame can be sent to the peer where the entry of the specific sending uniflow is left out. This could let to the peer unbinding the 4-tuple from its matching receiving uniflow.

Looking at the third design of C. Huitema, there also is no method provided for the active path manager to explicitly signal the loss of a specific IP-address. Instead, an endpoint that has lost the IP-address can implicitly signal its loss by sending a series of PATH_STATUS-frames for each of the affected paths to its peer carrying the path identifiers and a value that signals that the paths must be abandoned.

While IP-addresses can be lost at any point by an endpoint, other IP-addresses might become available to the endpoint later in the connection. The next section will explain how the design allows for sending uniflows to migrate to a different 4-tuple. Even unused uniflows that were used before (like sending uniflows 1 and 3 for the webserver) can be reused when new address pairs are found.

5.8 Migrating uniflows

One of the advantages that the current design of QUIC provides is the fact that endpoints are able to migrate a connection to a different 4-tuple whenever it is needed (for example, if one of the IP-addresses in the 4-tuple becomes unavailable to an endpoint). This allows for a connection to stay alive while providing a seamless handover experience to the user. As explained in section 4.2 an endpoint can initiate a migration by first performing a path validation procedure on a new 4-tuple. Here a new connection ID is chosen by each endpoint from the pool of available connection IDs that were communicated beforehand, to prevent linkability. If the path validation procedure succeeds, both endpoints can start to send data via the new 4-tuple.

This idea of migrating a connection to a different 4-tuple is also applied to the design of MultiPath QUIC. However, due to uniflows being used, the migration mechanism for MultiPath is altered slightly. First, both endpoints have to exchange additional uniflow-specific connection IDs to allow its peer to migrate a sending uniflow. If additional connection IDs are provided for the sending uniflow which the endpoint wants to migrate, a new 4-tuple can be identified. At this point, the same steps are performed as if the uniflow was a brand-new sending uniflow that has not been used before: The next step consists of choosing a new uniflow-specific connection ID and a path validation procedure where the endpoint checks if it can reach its peer via the 4-tuple. If this path validation procedure succeeds, the new 4-tuple is bound to the sending uniflow and the old 4-tuple is discarded. The peer will identify that the endpoint is trying to migrate a specific uniflow based on two reasons: The first reason being that it is able to identify the matching receiving uniflow via a previously unused uniflow-specific connection ID, and the second reason being that the 4-tuple of the 1RTT packet carrying a PATH_CHALLENGE-frame does not match the 4-tuple that was already bound to the receiving uniflow. If such a migration is detected by the peer, it can simply store the new 4-tuple in the context of the matching receiving uniflow. To complete the migration procedure, the endpoint that initiated the uniflow migration also sends a UNIFLOWS-frame to its peer via a 1RTT packet to inform the peer of the changes that were made.

At this point it is important to note that the context of each sending uniflow on an endpoint does carry a mechanism that regulates how much data can be sent over a path at any point in time. This mechanism, called the congestion controller, is path specific. Therefore, whenever a uniflow migrates to a different 4-tuple it also migrates to a different path. This path may have different transmission capabilities than the previous one, resulting in the need to reset the congestion controller to allow for better usage of the path's capabilities. In general, the context of a sending uniflow might contain additional performance metrics that are path specific, such as RTT estimators. Thus, resetting these performance metrics along with the congestion controller is referred to as resetting the context of a uniflow.

From the scenario's perspective: Some time after sending the REMOVE_ADDRESS-frame to the webserver, the laptop receives a 1RTT packet from the webserver. After identifying that its initial uniflow received the packet, the laptop decrypts the packet and sees that it contains a MP_NEW_CONNECTION_ID-frame, which holds a connection ID (0xef) for sending uniflow 0. After processing the frame, the laptop sends back a 1RTT packet to the webserver over sending uniflow 0, carrying an ACK-frame and marked with the destination connection ID of the server (0xbb). Afterwards, the laptop decides to migrate its initial sending uniflow from 4-tuple (A1 and B1) to a new 4-tuple that it identified (A2 and B2). First, the laptop performs a path validation procedure by generating a PATH_CHALLENGE-frame and placing a random value (0xdef) in it. The laptop then places the frame in a 1RTT packet, encrypts the packet and marks it with the webserver's new destination connection ID (0xef). The laptop then sends the packet to the webserver over sending uniflow 0.

Upon receiving the packets, the webserver identifies that its initial receiving uniflow received them. After decrypting the first packet the webserver identifies that the previously sent packet, containing the connection ID for uniflow 0 (0xef) has been successfully received by the laptop.

However, upon inspecting the second packet, the webserver notices that it is marked with a different connection ID (0xef), whereas the original connection ID was (0xbb). Additionally, the 4-tuple that is mentioned on the 1RTT packet (A1 and B1) also does not match the 4-tuple that is stored in the context of its initial receiving uniflow (A1 and B1). Based on this observation the webserver assumes that the peer is trying to migrate its initial sending uniflow. This assumption is confirmed when the webserver decrypts the packet and finds the PATH_CHALLENGE-frame. The observed 4-tuple is stored in the initial receiving uniflow's context, and its old 4-tuple is discarded. The challenge value is then extracted from the frame and placed in a new PATH_RESPONSE-frame. The webserver then places the frame in a 1RTT packet and encrypts the packet, marking it with the laptops destination connection ID (0xaa). Finally, the web-server sends the packet to the laptop over sending uniflow 0.

The laptop receives the packet and identifies that its initial receiving uniflow received it. The laptop then decrypts the packet and compares the challenge value that is extracted from the frame with the challenge value that was generated earlier. The laptop confirms that the two values are the same and can conclude that it can reach the webserver via the 4-tuple (A1 and B2). At this point the laptop wraps up the migration procedure for sending uniflow 0, by binding the new 4-tuple to the sending uniflow and resetting its context. Lastly, the laptop also generates a UNIFLOWS-frame to inform the webserver of the changes that were made. It places values 2 and 1 in the frame, indicating that the frame will contain two entries for the receiving uniflows, and one entry for a sending uniflow. For the first entry of the receiving uniflows the laptop places the values of 0 and 0 in the frame, indicating that its receiving uniflow 0 uses the IP-address that is marked with ID 0 (thus, A1) to receive packets. For the second entry of the receiving uniflows the laptop places the values of 2 and 0 in the frame, indicating that its receiving uniflow 2 uses the IP-address with ID 0 (thus, A1) to receive packets.

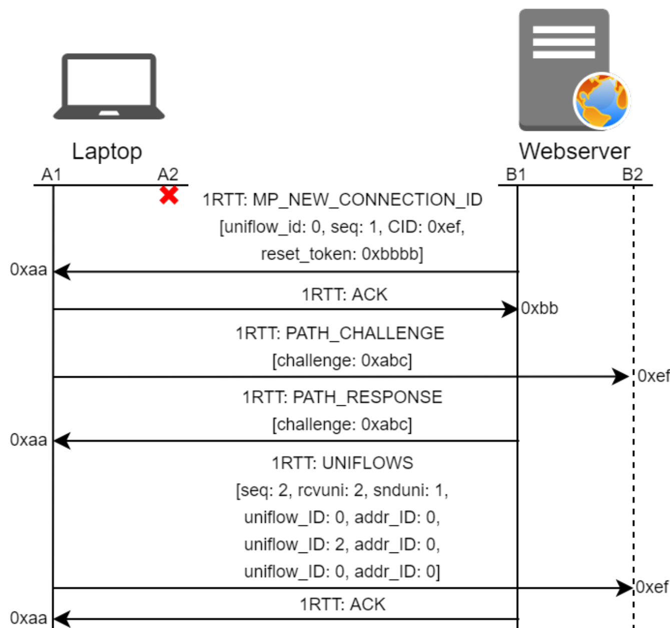


Figure 32: The laptop performing a migration of sending uniflow 0.

For the entry of the sending uniflow the laptop places the values of 0 and 0 in the frame, indicating that its sending uniflow 0 uses the IP-address with ID 0 (thus, B1) to send packets. The laptop then places the frame in a 1RTT packet and encrypts the packet, marking it with the webserver's destination connection ID (0xef). Finally, the laptop sends the packet to the webserver over sending uniflow 0. The webserver receives the packet and identifies that its initial receiving uniflow received it. The webserver then decrypts the packet and reads the content of the UNIFLOWS-frame. It concludes that no additional changes have been made. Finally, the webserver sends back a 1RTT packet to the laptop over sending uniflow 0, carrying an ACK-frame and marked with the destination connection ID of the laptop (0xaa). See Figure 32 for a visualisation.

The figure shows that the laptop's sending uniflow 0 has changed its connection ID halfway through the packet exchanges. This indicates that the sending uniflow has started sending packets over a different 4-tuple. It is important to note that the sending uniflow 0 of the webserver has not changed its connection ID, nor the 4-tuple that it is currently bound to. This is due to the fact that, in this design of MultiPath QUIC, it is not required for the two initial uniflows to keep transmitting data over the same 4-tuple, whereas this is required for a regular QUIC connection.

While this example covered the migration of an already active sending uniflow, the steps that need to be performed are the same when an endpoint wants to reuse a sending uniflow, but marked as unused. The only difference here is that the context of the sending uniflow has already been reset, and therefore on additional reset is required upon completing the path validation procedure.

It is important to note that, while performing this path validation procedure, it is entirely possible for the procedure to fail. This can happen for various reasons, such as the packet carrying the PATH_RESPONSE-frame being dropped. If this happens, the context on both endpoints is restored to its original form. The peer that received the PATH_CHALLENGE-frame will identify that it receives packets from a previously known 4-tuple on the affected receiving uniflow, and revert its context back to it.

Comparing this design to the other designs that are mentioned in section 5.1, it can be stated that this design gives both endpoints the most freedom in migrating its sending uniflows. The second design of Q. An et al. also allows its sub-connections to migrate to a different 4-tuple, but this is done via a similar approach as explained in section 4.2. This means that in this design both endpoints have to migrate that sub-connection to the same 4-tuple (even though both endpoints may have a different perspective over that 4-tuple), meaning that both endpoints must check if it can reach its peer via the new 4-tuple. The third design by C. Huitema does not really have a comparable form of support for path migration. In this design an endpoint can just signal a new path that can be used via the PATH_IDENTIFICATION-frame and the PATH_STATUS-frame. After this is done, the endpoint is able to choose which available path it will use to send data to its peer.

5.9 Transporting data

Unlike MPTCP, the mechanisms introduced to transport data in MPQUIC are less complex, this is because the QUIC transport design already introduces handy mechanisms that can be used by the MultiPath extension without any modifications. Taking a look at TCP, its MultiPath design required the introduction of a new level of management, called the connection level, to split data from a single input stream and send the parts over the subflows, while also generating enough control information to reassemble the split data into a single, complete, and in-order output stream of data. With this new level of management, a series of changes were applied to the semantics of TCP. For example, the meaning of a regular TCP ACK was changed to only have an impact on the subflow that it was sent on, instead of having an impact on the entire connection. A new signal, called the DATA_ACK was introduced to carry this connection-level impact. In addition to these changes, the extension also introduced a set of mechanisms, such as checksums and fallback operations to ensure that data could be delivered through a series of middleboxes that could perform unexpected actions. An explanation about this operation can be found in section 3.7. In comparison, the mechanisms in QUIC already allow a sending endpoint to generate and provide enough control information for the receiver to correctly receive the data.

Instead of having a single input stream like TCP, the sender can gather input data over multiple streams from a sending parent application (such as a HTTP/3-supporting webserver). The data in each stream is then consumed in parts in accordance to a frame scheduling policy, and each part is placed inside a STREAM-frame. Afterwards these frames are placed in 1RTT packets and sent to the peer. This is where one of the key designs of QUIC is brought to the foreground, because each STREAM-frame is given a set of fields that: identify a specific stream (stream ID), indicate how many bytes of data the frame contains (length) and specify the location of the data on the stream (offset). This combination of fields allows for STREAM-frames to be idempotent, and independent of the 1RTT packet that carries them, meaning that the frames can be processed by the receiver in an out-of-order manner and still result in correct placement of the data in the output streams that push to the receiving parent application (such as a HTTP/3-supporting browser).

Each 1RTT packet that is sent also carries a monotonically increasing packet number, to allow a receiving endpoint to signal to its peer which packets have been received correctly via ACK-frames. In addition to this, QUIC also provides the receiver with a mechanism to limit the sender in how much stream data that it can send. This mechanism makes use of flow control frames, like MAX_STREAM_DATA-frames and MAX_DATA-frames to indicate how much data that the receiver is willing to receive. Just like TCP, the use of these "receive windows" is further expanded upon with a congestion controller to ensure that the connection plays fair and shares the available resources of middleboxes with other existing connections. A more in-depth discussion about "receive windows" and congestion controllers (in MultiPath) can be found in section 8.1.

To address the issue of middleboxes, the design of regular QUIC already has built-in protection against external modifications, because everything is encrypted via keys that are derived from a shared secret that was established during the cryptographic handshake phase of a connection.

When applying the use of these STREAM-frames carried by 1RTT packets to the idea of multiple uniflows, no additional level of management needs to be introduced, as was the case with MPTCP. This is because MPQUIC is able to use the same STREAM-frames as regular QUIC, and since the uniflow on which a 1RTT packet is sent is packet-level information, each frame can be sent regardless of the uniflow or the 1RTT packet carrying it. Even duplication of a STREAM-frame and sending both frames over different uniflows will result in the same location of the stream being filled up with the same data due to the frames being idempotent. Other types of frames that carry flow control information, like the stream receive window advertised by the MAX_STREAM_DATA-frame also remain unchanged when there are multiple sending uniflows. The only requirement that needs to be fulfilled is that there must be a packet scheduler in place on the sending endpoint that decides which packets will be sent over which uniflow. An in-depth discussion about the concept of packet scheduling can be found in section 8.2.

There still remains one more issue that needs to be handled. The possibility exists that packet-level reordering can happen at the receiver side when multiple sending uniflows all have different latencies tied to them. The connection ID of the packet can already be used to ensure that each uniflow has a monotonically increasing packet number. However, to ensure flexibility and to reduce ACK-block sizes, each uniflow must also contain its own packet number space in its context.

If each uniflow will use its own packet number space, an additional need arises for a receiving endpoint to acknowledge these uniflow-specific packet numbers to its peer. Looking at the design of regular QUIC, this problem is already solved when a connection only uses the two initial uniflows, because an ACK-frame can carry the packet numbers of packets that a receiver has processed successfully from the initial receiving uniflow. To enable the acknowledgement of packets on the other receiving uniflows the extension introduces a new MP_ACK-frame that has the same functionality and fields as the ACK-frame. In addition to all the fields that a regular ACK-frame carries, a new uniflow ID-field is added to the MP_ACK-frame to identify from which receiving uniflow the host acknowledges packets. This new frame, just like all the other new MPQUIC frames, is also independent of the uniflow over which it is transported, or the 1RTT packet carrying it.

Combining all this together, from the scenario's perspective: after performing all the previous operations, the laptop now has one sending uniflow established in a connection with the remote webserver. This uniflow (sending uniflow 0) is currently coupled to the IP-addresses A1 and B2, and sends packets marked with connection ID 0xef. The other uniflow (sending uniflow 1) has been marked as unused, due to the laptop losing its WiFi-connection (and thus, IP-address A2). The webserver on the other hand now has two sending uniflows established, the first uniflow (sending uniflow 0) is currently coupled to the IP-addresses A1 and B1, and sends packets marked with the connection ID 0xaa. The second uniflow (uniflow 2) is currently coupled to the IP-addresses A1 and B2, and sends packets marked with the connection ID 0xdd (this CID was given by the laptop to identify receiving uniflow 2, but no example has covered this). An example transmission of "Hello world"-data for stream 0 can be seen in Figure 33. In this scenario the webserver decides to split Hello world into four parts: "He", "llo ", "wor" and "ld".

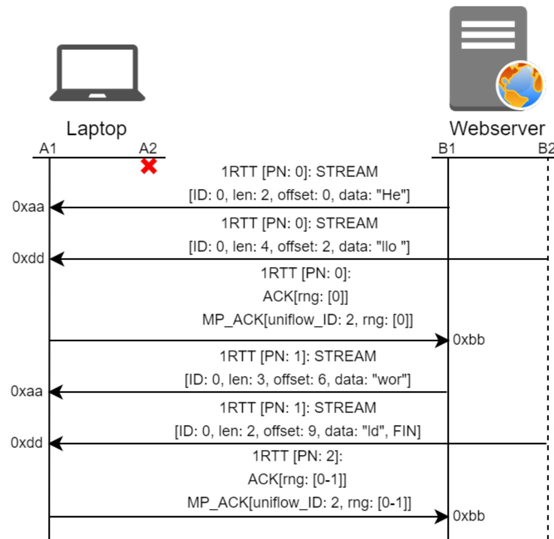


Figure 33: Transporting Hello world from the webserver to the laptop.

The webserver decides to send the first part of the data ("He") to the laptop via sending uniflow 0, while the second part of the data ("llo ") will be sent via sending uniflow 2. The webserver generates two STREAM-frames and places each part in a frame. For the frame with the first part the webserver places the values of 0, 2 and 0 in the frame, indicating that the frame carries 2 bytes of data for stream 0 and that the data should be placed at the beginning of the stream. For the second frame the webserver places the values 0, 4 and 2 in the frame, indicating that the frame carries 4 bytes of data for stream 0 as well and that it should be placed 2 bytes from the beginning of the stream. After this, the webserver places the first STREAM-frame in a 1RTT packet, encrypts the packet, and marks it with the destination connection ID of the laptop (0xaa). The webserver then sends the packet to the laptop over sending uniflow 0. After sending the first packet the webserver also places the second STREAM-frame in a 1RTT packet, encrypts the packet and marks it with the destination connection ID of the laptop (0xdd). Finally, the laptop also sends the second packet to the laptop via sending uniflow 2.

The laptop receives the two packets and identifies that its receiving uniflow 0 received the first packet, whilst the second packet was received on receiving uniflow 2. The laptop then decrypts both packets and reads the contents of the two stream frames. Upon inspecting the contents, it identifies that a new stream with ID 0 has been initiated and proceeds to reserve space to store the stream's data. After this, the laptop places the contents of the first frame at the beginning of the stream, while it places the contents of the second frame 2 bytes from the beginning of the stream, resulting in the following output stream 0: "Hello ". After processing both packets the laptop wants to acknowledge their reception to the webserver. Therefore, it generates both a ACK-frame and a MP_ACK-frame. In the ACK-frame the laptop places the range [0] to indicate that it received the first packet for receiving uniflow 0. In the MP_ACK-frame the laptop also places the range 0] to indicate that it received the first packet, but in addition the value of 2 is also placed in the frame, to indicate that the packet was received on receiving uniflow 2.

The laptop then places both frames in a 1RTT packet, encrypts the packet, and marks it with the destination connection ID of the webserver (0xef). Finally, the laptop sends the packet to the webserver over sending uniflow 0.

The webserver receives the packet and identifies that its receiving uniflow 0 received the packet. The webserver then decrypts the packet and reads the content of the ACK-frame and the MP_ACK-frame. Based on the information the webserver can conclude that the laptop has received both the first packet that was sent over the initial sending uniflow, as well as the second packet that was sent over sending uniflow 2. To proceed with the data transfer the webserver decides to send the third part of the data ("wor") to the laptop via sending uniflow 0, while the last part of the data ("ld") will be sent via sending uniflow 2. The webserver again generates two STREAM-frames and places each part in a frame. For the frame with the third part, the webserver places the values of 0, 3 and 6 in the frame, indicating that the frame carries 3 bytes of data for stream 0 and that the data should be placed 6 bytes from the beginning of the stream. For the second frame the webserver places the values 0, 2 and 9 in the frame, indicating that the frame carries 2 bytes of data for stream 0 as well and that it should be placed 9 bytes from the beginning of the stream. The webserver also enables the FIN-flag in this frame, to signal that all data for stream 0 has been sent. After this, the webserver places the first STREAM-frame in a 1RTT packet, encrypts the packet, and marks it with the destination connection ID of the laptop (0xaa). The webserver then sends the packet to the laptop over sending uniflow 0. After sending the third packet the webserver also places the second STREAM-frame in a 1RTT packet, encrypts the packet and marks it with the destination connection ID of the laptop (0xdd). Finally, the laptop also sends the fourth packet to the laptop via sending uniflow 2.

The laptop receives the two packets and identifies that its receiving uniflow 0 received the first packet, whilst the second packet was received on receiving uniflow 2. The laptop then decrypts both packets and reads the contents of the two stream frames. After this, the laptop places the contents of the first frame 6 bytes from the beginning of the stream, and it places the contents of the second frame 9 bytes from the beginning of the stream, resulting in the following output stream 0: "Hello world". The laptop also identifies that the FIN-flag was enabled in the second STREAM-frame, indicating that all data has been sent by the webserver. The laptop is now able to send the stream data to the parent application. After processing both packets the laptop wants to acknowledge their reception to the webserver. Therefore, it generates both a ACK-frame and a MP_ACK-frame. In the ACK-frame the laptop places the range [0-1] to indicate that it received the first and second packet for receiving uniflow 0. In the MP_ACK-frame the laptop also places the range [0-1] to indicate that it received the first two packets, but in addition the value of 2 is also placed in the frame, to indicate that the packets were received on receiving uniflow 2. The laptop then places both frames in a 1RTT packet, encrypts the packet, and marks it with the destination connection ID of the webserver (0xef). Finally, the laptop sends the packet to the webserver over sending uniflow 0.

The webserver receives the packet and identifies that its receiving uniflow 0 received the packet. The webserver then decrypts the packet and reads the content of the ACK-frame and the MP_ACK-frame. Based on the information the webserver can conclude that the laptop has received all the data for stream 0, finalizing the data transfer.

Comparing this design with the other designs mentioned in section 5.1, all three designs will be able to use the already existing mechanisms of QUIC. The independence of the STREAM-frame from both the uniflow and the 1RTT packet that carries it, is a key feature of QUIC that allows the MultiPath designs to operate without having to add an additional layer of management to the protocol. In the second design of Q. An et al., it is also proposed to use a new MP_ACK frame to acknowledge the packets on additional sub-connections, similar to the design of Q. De Coninck. In this case the Sub Connection Identifier is used instead of a uniflow ID to identify to which sub connection the acknowledged packet numbers belong to. For the third design of C. Huitema there is no additional frame that is equivalent to the MP_ACK-frame, because no additional concepts like uniflows or sub-connections are introduced. This leads to packets, that are being sent over the different paths, all sharing the same packet number space, and therefore they can be acknowledged by the already existing ACK-frames.

5.10 Closing a MPQUIC connection

Once all data has been communicated between the two endpoints and neither endpoint has anything left to send, the connection can be closed. For all three designs of MultiPath QUIC, closing a connection is exactly the same as in regular QUIC. The closure of a connection is initiated when either endpoint sends a CONNECTION_CLOSE-frame to its peer. After this, the endpoint enters the closing state to ensure that delayed or reordered packets are discarded properly, where it only keeps minimal connection state information, such as keys, connection IDs and version to handle incoming packets and respond with other CONNECTION_CLOSE-frames. Once the peer receives the CONNECTION_CLOSE-frame in a 1RTT packet, it enters the draining state after optionally responding with another CONNECTION_CLOSE-frame.

Just like in regular QUIC, an endpoint might detect that an error has occurred while performing MultiPath operations that leads to the connection having to be terminated. In this case the exact same procedure is followed as in regular QUIC, where the endpoint sends a CLOSE_CONNECTION-frame to its peer. The frame then also carries an error code and possibly a reason phrase for the connection closure.

From the scenario's perspective: After receiving the second 1RTT packet from the client and reading the (MP_)ACK-frames, both the webserver and the laptop have confirmed that all data has been sent over stream 0. Thus, the laptop decides to close the connection and enters the closing state. In this state the laptop creates a CONNECTION_CLOSE-frame, wherein it places the error code value of 0 to indicate that no errors have occurred, and that the connection can be closed immediately. After this the laptop places the frame in a 1RTT packet, encrypts the packet, and marks it with the destination connection ID of the webserver (0xef). Finally, the laptop sends the packet to the webserver via sending uniflow 0.

The webserver received the packet and identifies that its initial receiving uniflows received it. The webserver then decrypts the packet and reads the CONNECTION_CLOSE-frame. The webserver then also creates a CONNECTION_CLOSE-frame, and places the error code value of 0 to indicate that no errors have occurred, and that it agrees to also close the connection immediately.

After this the webserver places the frame in a 1RTT packet, encrypts the packet, and marks it with the destination connection ID of the laptop (0xdd). Finally, the webserver sends the packet to the laptop via sending uniflow 1. (This packet can also be sent over either sending uniflow.) After sending the packet the webserver enters the draining state. From this point on both the laptop and the webserver will remain in their states for a predefined amount of time, to ensure that all reordered or delayed packets have been processed. After this both endpoints discard the remaining state of the connection. See Figure 34 for a visualisation.

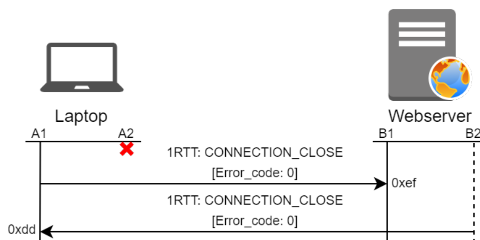


Figure 34: Closing a MPQUIC connection.

5.11 Problems of MPQUIC

While MultiPath QUIC introduces the concept of uniflows in such a way that it uses the many already existing mechanisms in QUIC to prevent the introduction of new problems as much as possible, there still are some problems that remain. The first problem is the fact that the extension slightly alters the semantics of a path validation procedure. Under normal circumstances, since draft-32 of the QUIC transport protocol, it is expected that a PATH_CHALLENGE-frame is received by the path validation initiator on the same IP-address that it has sent the PATH_CHALLENGE-frame from. This, in combination with the fact that the packets carrying the frames must be padded to a minimum size of 1200 bytes, ensures that path validation only succeeds if the path is functional in both directions. The minimum of 1200 bytes is given to prevent DoS amplification attacks. However, by transitioning from a bidirectional path to uniflows that use a given path in one direction, there is no need for this bidirectional insurance. It is only required that the sender can reach the peer over the uniflow that is bound to a certain path, because the peer might have multiple sending uniflows that use different paths to reach the sender and acknowledge the packets.

Additionally, due to this design still being in early draft versions, there is one small part that is proposed in the draft but is seemingly not of any use for an implementation. After any changes to a uniflow's 4-tuple (due to the addition of an IP-address, migration of a sending uniflow or loss of an IP-address), the affected endpoint will send a UNIFLOWS-frame to its peer to inform the peer about these changes. In response, the peer can also send back a UNIFLOWS-frame to inform about the changes that it had to make. This UNIFLOWS-frame contains two categories of info sections, one for the endpoint's receiving uniflows and their used local address IDs, and another for the endpoint's active sending uniflows and their used local address IDs. However, only the category that contains the info of a peer's sending uniflows is useful to the receiver of the frame. A discussion about why the receiving uniflows info section can be removed can be found in section 8.4.2.

Moving the perspective to the MultiPath extension in general, there is the fact that multiple design proposals exist for the extension in QUIC (see section 5.1). It is obvious that the design of Q. De Coninck is the most complex and comprehensive form of MP-operations in QUIC. However, only one implementation currently supports the design. In second place comes the design of Q. An et al, which also introduces a new concept, called sub-connections to the protocol. However, the design is simpler that the first one, as no mechanisms, like uniflow-specific connection IDs and UNIFLOWS-frames are used. This design is already used in the Taobao mobile app [24]. In third place comes the design of C. Huitema, who only introduces an active path management system to enable MultiPath. At the time of writing this thesis, the last two design drafts are not yet fully complete (there are still mechanisms that need to be explained in their respective drafts). So, no real argument can yet be made about which design is the most suitable for QUIC. However, there are some questions that still can be asked. For example, is it desirable for uniflows to be used in comparison to bidirectional paths? How much fine-grained control is really needed? How large is the number of intermediate links that only support unidirectional communication, such as non-broadcast satellite links? What are the security risks of each design (This has yet to be determined for the design of C. Huitema)?

5.12 Differences between MPTCP and MPQUIC

After inspecting both MultiPath designs, there are several differences that can be identified between the TCP design and the QUIC design of Q. De Coninck. In this section an overview of the most important differences is given.

First, it can be identified that MPTCP and MPQUIC both have a different perspective on what using multiple paths simultaneously actually means. Regular TCP ensures reliable data delivery by sending back acknowledgments to the sender. Due to the regular design only accounting for having a single path between a sender and a receiver, the data flows in one direction and acknowledgments in the other on that same path. Therefore, the notion of bidirectional paths was established. The MPTCP design just copied the idea of this bidirectional path and applied it multiple times to support the concept of MultiPath. The MultiPath QUIC design on the other has identified that this bidirectional path actually exists out of two independent packet flows that coincidentally use the same path. More specific, it is perfectly possible for the packet flow from the server to the client to follow a different physical route in the network than the packet flow from then client to the server. Thus, QUIC introduced uniflows that use a given path in one direction to send data from one endpoint to the other. A different uniflow (that can use a different path) is then used by the peer in the other direction to acknowledge the data that it received from the sender.

Second, both designs have to negotiate support for the MultiPath extension. However, MPTCP relies on the TCP options field that is located in the header of a TCP packet. During the three-way handshake phase both endpoints send a clear-text MP_CAPABLE-option to exchange keys. These keys are used later in the connection for authentication of additional subflows and identification of a connection. In MPQUIC on the other hand, endpoints use authenticated transport parameters that are exchanged during the handshake phase of the connection to signal MP support.

Third, after connection establishment MPTCP can initiate additional subflows by performing another TCP handshake with the peer over a different set of IP-addresses. In that handshake it sends a clear-text MP_JOIN-option that carries a token and an HMAC-message that is derived from the previously exchanged keys. If the handshake completes the new subflow can be used to transport data to the peer. MPTCP on the other hand uses a path validation procedure in combination with the exchange of uniflow specific connection IDs to initiate and use additional uniflows. If the path validation procedure succeeds the uniflow can be used to send data to the peer. Additionally, MPQUIC also provides both endpoints with the ability to limit how many uniflows its peer can use to send data by limiting which uniflows are identified via a uniflow-specific connection ID, whereas MPTCP endpoints must reject the handshake attempt after the peer has sent it.

Fourth, both MPTCP and MPQUIC provide two ways to signal additional IP-addresses to their peer. However, the explicit method in MPTCP introduces a couple of issues, the first one being the fact that these IP-addresses are sent in clear-text via the ADD_ADDRESS option, which results in the senders IP-address being leaked to the network. The second problem is that there is no acknowledgement mechanism to ensure the options reception by the peer. (The regular TCP ack of the packet carrying the option is not enough, because the option can be stripped by middleboxes.) MPQUIC on the other hand does not have these issues, because the frames that carry these IP-addresses are always encrypted, and the frames are carried by packets for which the sender can receive acknowledgements from the peer.

Lastly, once additional IP-addresses have been exchanged and additional subflows have been established, MPTCP must make sure that these subflow all behave like regular TCP connections to prevent middlebox interference as much as possible. This meant that, to ensure that packet reordering does not introduce new problems, a new level of management had to be introduced to TCP where an additional Data Sequence Number was used. Each subflow now had to deal with the acknowledgement of its own sequence numbers as well as the signalling of a mapping and the DATA_ACKs of Data Sequence Numbers. MPQUIC on the other hand did not have to introduce a new level of management to the protocol, since no middleboxes could alter the data that is being sent over the multiple uniflows. Additionally, STREAM-frames are independent of the packet carrying them, and therefore can be sent over any uniflow. On top of that, the frames were also idempotent, meaning that duplication or retransmission over a different uniflow was possible, without having to worry about data being misplaced on the receiver side. In comparison, MPTCP had to send in-sequence data over a uniflow so that the mapping would remain correct. Additionally, retransmission of data over a different subflow still meant that the original transmission of the data on the original subflow still had to be acknowledged, to ensure that subflows kept on behaving like regular TCP connections.

6 An MPQUIC implementation

After inspecting the proposed mechanisms in the draft of Q. De Coninck et al., the next step would be to implement and evaluate those mechanisms. To implement the concept of MultiPath as an extension for QUIC, multiple options can be pursued. A first option could be to create a new QUIC implementation that we further expand with the mechanisms from the draft. However, this approach would require a lot of time to test and evaluate the default mechanisms of QUIC. Thus, a second option was chosen. Here, we implement the MultiPath extension in an open-source QUIC implementation. Since there are multiple open-source implementations available, the one that best fits our needs can be chosen. For this thesis, the AIOQUIC implementation [25] was chosen.

6.1 Aioquic

AIOQUIC is a Python implementation of the QUIC protocol. The implementation is written by Jeremy Lainé, and it features a minimal TLS 1.3 implementation, along a QUIC stack and an HTTP/3 stack. At the time of writing this thesis, the implementation closely tracks the QUIC specification drafts and is regularly tested for interop against other implementations. We chose this implementation because it is a well documented implementation that allows us to write the mechanisms of the draft in an effective manner. Additionally, the implementation is being tested constantly via a public interop runner that evaluates connections between multiple implementations. Besides this, the implementation also provides its own testing mechanisms that can be expanded upon to include tests for our MultiPath implementation.

6.1.1 Asyncio and SansIO

AIOQUIC uses the Asyncio library to allow the implementation to operate in an event-based manner. Whenever a packet arrives at an endpoint that runs the AIOQUIC implementation, an event occurs that starts mechanisms which process the received packet. The implementation then performs a series of actions in accordance to the QUIC transport draft. The implementation uses the DatagramProtocol-class to set up an endpoint. Besides this, the implementation follows the SansIO pattern [26], where actual I/O operations are left to the user of the implementation.

6.1.2 Structure

The AIOQUIC implementation provides a series of classes in a hierarchical order. On the top level, the example http3_server.py and http3_client.py files provide an entrypoint to the implementation. Both entrypoints take a series of input parameters that allow for the behaviour of the implementation to be configured. Examples of input parameters are, but not limited to, the IP-address and port of the server, the use of specified certificates, the URL for which data should be requested, and the directory to which logging information needs to be written. From these example files, a server or client endpoint is created. This is done via the DatagramProtocol-class from the Asyncio library. This class (from server.py or client.py) allows for an endpoint to be set up at an available IP-address and port combination. The endpoint can then be used to receive and send packets. Additionally, a protocol class can be given to an object from the DatagramProtocol-class. This protocol class defines the behavior of the endpoint when, for example, it receives a new packet.

In the case of the client, this protocol class is a QuicConnectionProtocol-class that keeps track of the transport that allows for sending packets, along with a QuicConnection-object that keeps track of the connection's state. In this class, received packets are processed, the connection's state is updated and packets are prepared for sending in accordance to mechanisms in the QUIC transport draft.

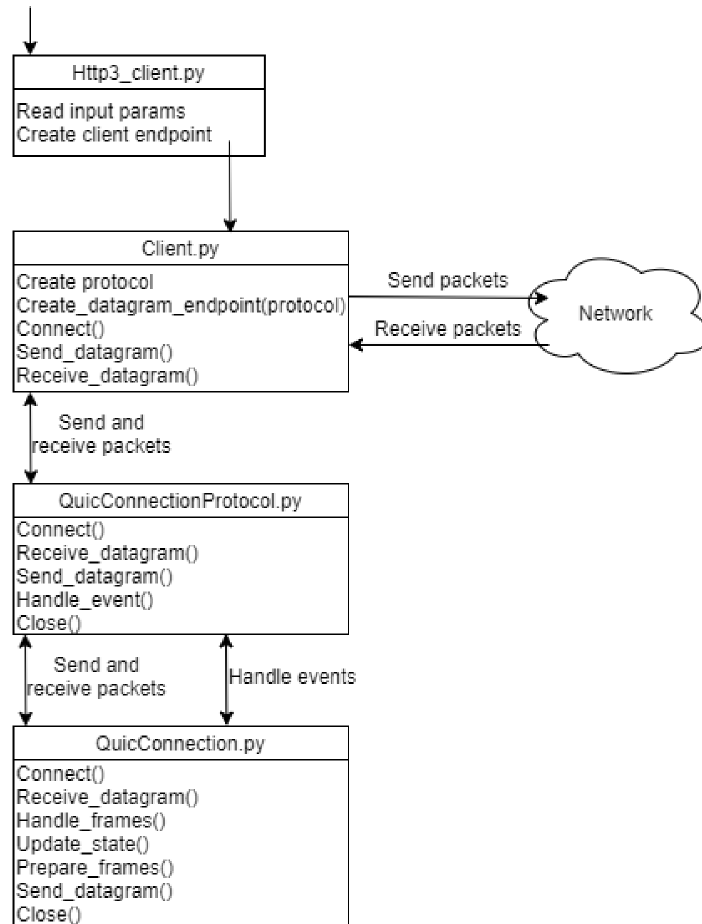


Figure 35: An AIOQUIC client.

A broadly explained example: if a client receives a packet from an established connection, the packet will arrive at the endpoint created in Client.py. This endpoint will pass the packet to the QuicConnectionProtocol-object in Protocol.py, to allow correct endpoint behaviour. From here, the packet is further passed on to the QuicConnection-object in Connection.py. The QuicConnection-object will process the packet and modifies its state in accordance to the mechanisms of the transport draft. Events that occurred during the processing are passed back to the QuicConnectionProtocol-object to allow for further actions to be taken, such as storing requested file data. The QuicConnection-object also prepares packets that are to be sent in response to the received packet. These packets are passed back to the QuicConnectionProtocol-object, which in turn passes the packets to the endpoint. Finally, the endpoint will sent the packets to the peer. See Figure 35 for a visualisation.

In case of the server, an additional protocol layer is placed between the endpoint and the QuicConnectionProtocol-class. This is because the server can support multiple connections at the same time, each having their own QuicConnectionProtocol-object (and QuicConnection-object). Thus, the additional QuicServer protocol-object has the responsibility of delegating the received packets to the correct connection, represented by their QuicConnectionProtocol-object. If the packet indicates a new connection attempt, an additional connection (and QuicConnectionProtocol-object) is created by the QuicServer protocol-object.

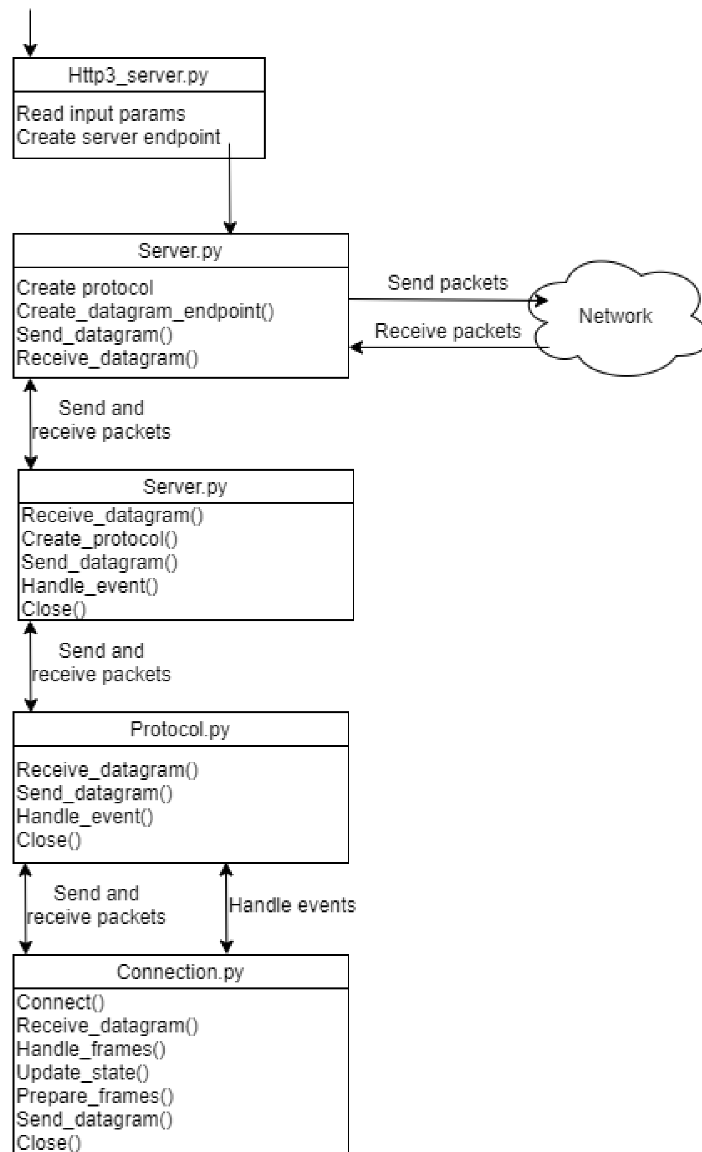


Figure 36: An AIOQUIC server.

A broadly explained example: if a server receives a packet from an established connection, the packet will arrive at the endpoint created in Server.py. This endpoint will pass the packet to the QuicServer-object in Server.py, to allow correct endpoint behaviour. The QuicServer-object will inspect the packet's connection ID and search the already existing connections for a match. If no match is found and the packet indicates a new connection attempt, a new connection (and a QuicConnectionProtocol-object) is created. If a match is found the packet is further passed on to the matching QuicConnectionProtocol-object. From here, the packet is further passed on to the QuicConnection-object in Connection.py. The QuicConnection-object will process the packet and modifies its state in accordance to the mechanisms of the transport draft. Events that occurred during the processing are passed back to the QuicConnectionProtocol-object and the QuicServer-object to allow for further actions to be taken, such as providing requested file data. The QuicConnection-object also prepares packets that are to be sent in response to the received packet. These packets are passed back to the QuicConnectionProtocol-object, which in turn passes the packets to the endpoint. Finally, the endpoint will sent the packets to the peer. See Figure 36 for a visualisation.

6.2 Implementing MultiPath

To implement the concept of MultiPath in AIOQUIC, a series of steps have to be taken. First, an endpoint (thus, both client and server) must be allowed to have multiple local IP-address and port combinations available (section 6.2.1). Second, a MPQUIC connection must be aware of these additional local IP-addresses (section 6.2.2). Third, a MPQUIC connection should be expanded to allow the transmission of new MP-specific frames (section 6.2.3). Fourth, a MPQUIC connection should be aware of local sending and receiving uniflows (section 6.2.4). Finally, the sending uniflows in a connection should each be able to send data via additionally established paths (section 6.2.5).

6.2.1 Implementing multiple datagram endpoints

The first step to implement MultiPath in AIOQUIC is to enable multiple IP-addresses to be used by QUIC connections for the sending and receiving of packets. From the explanation in section 6.1.2, both the client and the server enable the sending and receiving of packets on a given IP-address via the DatagramProtocol-class from the Asyncio library. Thus, a good solution to allow the use of multiple IP-addresses to be used is to create multiple DatagramProtocol-endpoints. But before this can be done, it should be possible for the implementation to bind endpoints to IP-addresses that the developer has chosen, in a dynamic way. For this to be implemented, additional input parameters can be defined. For the server, the input parameters were extended by introducing a "ports"-parameter and a "multipath"-parameter. The first parameter allows for a series of ports to be passed to the implementation, while the second parameter allows for a select maximum number of sending uniflows to be created during the connection. For the client, four additional parameters are introduced, being "local-host", "local-ports", "local-preferred-port" and "multipath". The first parameter allows for the endpoint to be bound to a different IP-address than "::1". The second parameter also allows for a series of ports to be passed to the implementation. The third parameter allows for specification from which port the connection to a server should be initiated. Lastly, the fourth parameter also allows for a select maximum number of sending uniflows to be created during the connection.

For each of the ports listed in the input parameter, both the server and the client will create a DatagramProtocol-endpoint and bind it to the given IP-address and port. However, there are additional changes that must be implemented before connections can use any of these endpoints to send and receive packets. In case of the server, these endpoints must share the list of existing connections between themselves. This can be done by providing a shared dictionary to these endpoint. Additionally, the transport objects from these endpoints must also be given to the connections to allow them to send packets into the network, thus another dictionary is given. See listing 1 for a visualisation.

```
protocols: Dict[bytes, QuicConnectionProtocol] = {}
transports: Dict[str, asyncio.DatagramTransport] = {}
loop = asyncio.get_event_loop()
for port in ports:
    loop.run_until_complete(
        serve(
            args.host,
            port,
            configuration=configuration,
            create_protocol=HttpServerProtocol,
            protocols=protocols,
            transports=transports,
            session_ticket_fetcher=ticket_store.pop,
            session_ticket_handler=ticket_store.add,
            retry=args.retry,
        )
    )
```

Listing 1: Creating multiple endpoints.

In case of the client, an additional protocol layer was chosen to be placed between the endpoint and the QuicConnectionProtocol-class. In this additional QuicClient-protocol a similar mechanism to that of a regular server is implemented. The only difference here is that only a single connection has to be tracked, whereas a server would have to keep track of multiple connections. The decision, to convert the client from being a single endpoint that directly follows the QuicConnectionProtocol-class towards a set of endpoints that follow the QuicClient-protocol which manages a QuicConnectionProtocol-object, was made to simplify the complexity of a client operation and make its operations similar to that of a server.

6.2.2 Introducing additional IP-addresses to the connection

The second step to implement MultiPath in AIOQUIC is to introduce the additional IP-addresses to the QuicConnection-class. There already is a way to introduce connection-specific information from the entrypoint of the implementation to the QuicConnection-class via the endpoint's protocol. A Configuration-object is created and given to the protocol that dictates the behaviour of the endpoint. From here, the Configuration-object is passed to the QuicConnection-class upon creating a new connection. See listing 2 for a visualisation.

```
# prepare QUIC connection
connection = QuicConnection(configuration=configuration, ...)
```

Listing 2: Passing a configuration to the QuicConnection-object.

Simply expanding this Configuration-object allows for the IP-addresses to be introduced in the QuicConnection-class. However, these IP-addresses also had to be tracked by the QuicConnection-class. There already was a QuicNetworkPath-class available that allowed a regular QUIC connection to keep track of remote IP-addresses. However, this class did not meet the requirements of tracking MultiPath-related IP-addresses. Thus, an EndpointAddress-class was introduced. that allowed the QuicConnection-class to keep track of both local and remote IP-addresses (and their information). See listing 3 for a visualisation.

```
self._local_addresses: Dict[int, EndpointAddress] = {}
# Copy the addresses shared in the configuration
for i in range(len(configuration.local_addresses)):
    laddr = configuration.local_addresses[i]
    addr = EndpointAddress(
        address_id=i,
        ip_version=laddr[1],
        interface_type=laddr[2],
        ip_address=laddr[0],
        port=(laddr[3] if len(laddr) >= 4 else None),
        sequence_number=0,
    )
    self._local_addresses[i] = addr

self._remote_addresses: Dict[int, EndpointAddress] = {}
```

Listing 3: Tracking the local and remote IP-addresses.

6.2.3 Implementing the handling and writing of MP-specific frames

The third step to implement MultiPath in AIOQUIC is to allow the connection to process and write MP-specific frames. In this step, we can also rely on the already existing mechanisms of AIOQUIC, because a series of handle_<frame_type>_frame, write_<frame_type>_frame and on_<frame_type>_delivery-functions have already been defined. Each of these functions is "dynamically" called with additional function-parameter when a packet is processed, is prepared, or signaled as delivered/lost.

Thus, implementing the handling and writing of MP-specific frames was simply creating additional handle-, write-, and delivery-functions for each MP-frame, where each function took the same function-parameters as the others and returned the same type of results. Additionally, the MP-frames also had to be added to the __frame_handlers-object. Lastly, the writing of these frames also had to be included in the write_application-function, to allows these frames to be created when needed. See listing 4 for a visualisation.

```
# frame handlers
self.\_\_frame\_handlers = {
    <other, regular QUIC frames>,
    0x40: (self._handle_mp_new_connection_id_frame, EPOCHS("1")),
    0x41: (self._handle_mp_retire_connection_id_frame, EPOCHS("1")),
    0x42: (self._handle_mp_ack_frame, EPOCHS("1")),
    0x43: (self._handle_mp_ack_frame, EPOCHS("1")),
    0x44: (self._handle_add_address_frame, EPOCHS("1")),
    0x45: (self._handle_remove_address_frame, EPOCHS("1")),
    0x46: (self._handle_uniflows_frame, EPOCHS("1")),
}
```

Listing 4: QuicConnection-class's frame handlers.

6.2.4 Implementing uniflow classes

```
class QuicReceivingUniflow:
    self.uniflow_id: int
    self.cid_available: List[QuicConnectionId]
    self.cid: bytes
    self.cid_seq: int
    self.packet_number: int

    self.source_address: Optional[EndpointAddress]
    self.destination_address: Optional[EndpointAddress]

    self.receiving_spaces: Dict[tls.Epoch, QuicReceivingPacketSpace]

class QuicSendingUniflow:
    self.configuration: QuicConfiguration
    self.uniflow_id: int = uniflow_id
    self.cid_available: List[QuicConnectionId]
    self.packet_number: int

    self.state: UniflowState

    self.source_address: Optional[EndpointAddress]
    self.destination_address: Optional[EndpointAddress]

    # Congestion controller
    # Performance metrics
    self.sending_spaces: Dict[tls.Epoch, QuicSendingPacketSpace]
    self.loss: QuicPacketRecovery
```

Listing 5: QuicConnection-class's sending and receiving uniflow classes.

The fourth step to implement MultiPath in AIOQUIC is to make the QuicConnection-class aware of the multiple available sending and receiving uniflows. In this step, two new classes are introduced, being the QuicReceivingUniflow-class and the QuicSendingUniflow-class. Both these classes contain the required information that was proposed in the draft text, such as, but not limited to, the uniflow's ID, the set of connection IDs that belong to a given uniflow, the packet number of that uniflow and the source and destination IP-addresses that the uniflow is temporarily bound to. Additionally, each receiving uniflow also tracks its own receiving_space, whereas each sending uniflow also tracks its sending_space and the state of its own congestion controller. See listing 5 for a visualisation.

Initially, the QuicConnection-class will only track a single QuicSendingUniflow-object and QuicReceivingUniflow-object, as stated by the mechanisms in the draft. After the handshake phase of the connection has been completed, additional sending and receiving uniflow objects are created and tracked by the QuicConnection-class.

6.2.5 Implementing the ability to send and receive data for multiple uniflows

The fifth step to implement MultiPath in AIOQUIC is to allow the additional sending and receiving uniflows to respectively send and receive data in a given connection. The AIOQUIC implementation already provides methods in the QuicConnection-class to allow the reception and creation of packets. Further expanding upon these methods would allow MultiPath operations to be made possible.

First, the receive_datagrams-function is altered to allow the identification of the receiving uniflow based on the receiving packet's connection ID. After identifying a receiving uniflow, the packet is processed in accordance to the mechanisms in the QUIC transport draft. The state of the connection, along with the states of the sending and receiving uniflows is altered during this process. Additionally, the IP-address information of the packet is checked to identify if the peer has migrated to a different 4-tuple. If this is the case, the IP-address information of that receiving uniflow is updated. Finally, the packet is also registered in the receiving uniflow's receive_space.

Second, the datagrams_to_send-function is also altered. Here, the sending uniflows are split into two categories, one contains sending uniflows that are not yet bound to a validated 4-tuple, while the other contains the sending uniflows that have a connection ID and are bound to a validated 4-tuple. After this, a check is performed if the connection has to be closed. If this is the case, a packet is generated to signal this action. In the case that the connection remains open, each of the sending uniflows from the first category are bound to a new 4-tuple, and a path_validation procedure is started. This will allow additional sending uniflows to be initiated and to later be used to send actual data to the peer. Lastly, for each of the sending uniflows in the second connection, a packet-builder that will allow frames to be placed in packets for each of the sending uniflows is provided. These packet builders are then passed to the write_application-function. In the write_application-function, each builder, along with additional uniflow information is placed in a BuilderManager-object.

From here, the write_application will generate the required control-frames, stream-frames and the MP-specific-frames. These frames are placed in the packet builder of a specific sending uniflow, in accordance to the BuilderManager-object. This BuilderManager-object provides mechanisms that allow packet scheduling to occur (explained in section 8.2). The write_application also takes into consideration the limits of the sending uniflow's builder, in accordance to the uniflow's congestion window limitations (explained in section 8.1).

6.3 Missing features

The current implementation of MultiPath in AIOQUIC has not yet implemented all the mechanisms that are proposed in the draft of Q. De Coninck et al. For example, the implementation only provides a dummy congestion controller and a dummy packet scheduler. The provided dummy implementations are placeholders that do not yet perform the required operations of a congestion controller or a packet scheduler. In the case of the congestion controller, the congestion window is kept at a constant size, being the size of the initial window. In the case of the packet scheduler, only a frame-based Round-Robin scheduler is provided that loops over the available packet builders in a circular fashion. A discussion of the implementation of a real congestion controller and packet scheduler is given in section 8.4.1. Besides this, there are additional features missing in the current implementation. For example, the current implementation does not allow for an endpoint to be closed or added, where IP-addresses could respectively be lost, of found.

6.4 Evaluation

To evaluate the implementation of the MultiPath concept in AIOQUIC, we can use the already provided test mechanisms. Here, an additional set of tests can be written where the different aspects of the MultiPath operations are tested. For example, a test can be written that checks if two endpoints can correctly exchange connection IDs that belong to specific sending uniflows. Additionally, the exchange of additional addresses can be tested as well. However, not all aspects of MultiPath are currently tested via this testing mechanism. For example, no tests are available that check whether an endpoint sends MP_ACK-frames in a timely and correct manner, nor are there any tests that validate if an endpoint initiates additional paths in a correct manner. While these tests must certainly be implemented in future versions, a partial validation can already be done in the form of interops with other implementations. See section 7.2 for an explanation.

7 Evaluating MPAIOQUIC against other implementations

After implementing the concept of MultiPath in MPAIOQUIC, the next step would be to evaluate the implementation set out in this thesis and compare it against other existing implementations. In the QUIC Working Group, there are a number of additional implementations that test and evaluate the proposed mechanisms of QUIC [27]. From this set of QUIC implementations, most of them are open-source projects that implement the base transport mechanisms of QUIC. There are some implementations that support additional features, such as HTTP/3 Over QUIC, QPACK, and other extensions such as MultiPath QUIC.

Since multiple implementations have been developed for QUIC, numerous steps have been taken to simplify the testing and validation of these implementations. For instance, a public automated interop runner has been made available to the Working Group [28]. The interop runner documents the current interop status of various QUIC client and server implementations that have chosen to participate in the automated testing. Currently, there are 15 participating implementations. To document the current interop status, a series of test cases are defined that each participating implementation has to provide interop code for. These test cases are based on the features that QUIC provides, such as, but not limited to, version negotiation, handshake completion, session resumption, 0-RTT and HTTP/3 data transfer. Via an interop matrix that is performed multiple times a day where different server and client implementations connect with each other, the current support or lack thereof for the test cases is visualized.

Besides the provision of a public interop runner, tools such as the QLOG logging schema and the QVIS visualisation tool have been developed to better visualise and understand how implementations are behaving in certain scenario's [29]. At the time of writing this thesis, many implementations have implemented logging mechanisms that make use of the QLOG logging schema, including AIOQUIC. Via this logging schema the implementations are able to log events that occur during connections in .qlog-files. These files can then be loaded into the QVIS visualisation tool to visualize how the connection behaved. Here, visualisations such as sequence diagrams, congestion graphs, stream graphs, packetization diagrams and other statistics are visualised.

7.1 MP Coverage

An initial idea to evaluate our implementation could have been to register our implementation as a participant of the public interop runner. However, this idea comes with a number of drawbacks. In general, there are a number of implementations that support the basic transport mechanisms of QUIC, and these mechanisms can be tested via the interop matrix. But because the MultiPath concept is currently only proposed in the form of an extension, there are not a lot of implementations that support MultiPath. Besides this, there are no tests defined in the interop that focus on the MP aspect. Making matters even worse, not every implementation that supports MP has implemented the same proposed concept. As discussed in section 5, there currently are multiple proposals that take a different perspective on implementing the MultiPath concept. These proposals are not compatible with each other, thus no interop can be performed between them. Luckily, there is one implementation that does implement the mechanisms from the draft of Q. De Coninck et al., being PQUIC [30].

7.1.1 PQUIC

PQUIC is a C implementation of the QUIC protocol that is based on the PICOQUIC implementation, and is developed by a research team at UCLouvain. The implementation aims to provide a framework that enables QUIC clients and servers to dynamically exchange protocol plugins that extend the protocol on a per-connection basis. From the paper [30], the argument for providing such a framework is based on the fact that application requirements can evolve over time, potentially requiring protocols to adapt. Such adaptations are traditionally done by negotiating protocol extension support during the handshake, and experience with TCP has shown that this leads to delays of several years or more to widely deploy standardized extensions. The framework would allow endpoints to exchange plugins with each other that could alter and/or extend the behaviour of the protocol on both endpoints. With this framework, a series of plugins have been provided, such as Forward Error Correction, QLOG support, ACK_Delay and MultiPath.

7.2 Performing Evaluations

Due to the initial idea of using the public interop runner being sub-optimal, a different approach must be taken to evaluate our implementation. Our approach exists of two phases, the first phase has the goal of performing interops between MPAIOQUIC and other implementations to validate that our implementation does not deviate from normal QUIC behaviour when a connection is made with a regular server or client. The second phase has the goal of performing interops between MPAIOQUIC and PQUIC to validate that the two implementations behave as envisioned by the proposed mechanisms of the draft.

7.2.1 Handling other implementations

To accomplish the goal of performing interops between MPAIOQUIC and other implementations, some form of interop code has to be provided, where the MPAIOQUIC-client can connect to the servers of other implementations. Luckily, because AIOQUIC already participates in the public interop runner, this interop code is already provided. In this code, the server addresses of each participating implementation have been defined, and a set of test functions have been provided where the tests, defined by the public interop, can be performed. In these tests functions a number of steps were taken. First, an AIOQUIC-client is created and instructed to connect to a selected server. After this, in the second step the specific test is performed. In the third step the connection is terminated, and the result of the test is returned.

Most of the provided code could be reused in MPAIOQUIC, however the NAT_rebinding and Address_mobility tests had to be altered to work with the changed MultiPath implementation of MPAIOQUIC. Originally, in AIOQUIC the changes of IP-addresses were simulated by closing the transport from which the packets were sent, after which a new transport was created that carried a different IP-address. See Listing 6 for a visualisation. In MPAIOQUIC this was changed, because the client already had multiple transports available. To simulate the IP-address changes, the source IP-address of the initial sending uniflow was changed, resulting in a different transport being used. See Listing 7 for a visualisation.

```python
# replace transport
protocol._transport.close()
await loop.create_datagram_endpoint(
    lambda: protocol, local_addr=("::", 0)
)
```

Listing 6: Changing IP-address of the client in AIOQUIC.

```python
# change the local address of the initial sending uniflow
siunuflow = protocol._quic._sending_uniflows[0]
siunuflow.source_address = protocol._quic._local_addresses[1]
```

Listing 7: Changing IP-address of the client in MPAIOQUIC.

From here, interops with other implementations were performed at different stages of the implementation of the MultiPath mechanisms in MPAIOQUIC. The results of the AIOQUIC interop can bee seen in Figure 37a, while the final results of the MPAIOQUIC interop can be seen in Figure 37b. The meaning of each letter in the summary table can be found in [31]. The most prominent difference between the two figures is the addition of the aioquicMP entry for the interop results of MPAIOQUIC, this is because an additional change was introduced here. Besides the list of available remote servers, an additional server entry was added that contains the IP-address of a local MPAIOQUIC-server. This addition allowed us to partially validate that the behaviour of both the MPAIOQUIC-server and client is correct, in accordance to the interop tests.

```
SUMMARY
akamaiquic          VH-CR--Q M------- ---
aioquic             VHDCRZSQ MBAUP-L- 3dp
ats                 VHDCRZSQ MBA----- ---
f5                  VHDCRZ-Q M--UP--- 3d-
haskell             VHDCRZSQ MBA---L- 3--
gquic               VHDCRZ-Q M------- 3d-
lsquic              VHDC--SQ MBAUP--- 3dp
msquic              VHDCRZSQ MBAUP-L- ---
mvfst               VHDCRZ-Q MBA---L- 3dp
ngtcp2              VHDCRZSQ MBAU--L- 3dp
ngx_quic            VHDC---Q M------- 3--
pandora             -------- -------- ---
picoquic            -------- ------L- ---
quant               VH-CRZ-Q MBAU--L- ---
quic-go             VHDCRZSQ M--U---- 3--
quiche              VHDCRZSQ M------- 3--
quicly              --D----- -------- 3--
quinn               -------- -------- ---
```

(a) Interop results of AIOQUIC.

```
SUMMARY
akamaiquic          -------- -------- ---
aioquic             VHDCRZSQ MBAUP-L- 3dp
ats                 VHDCRZSQ MBA----- ---
f5                  VHDCRZ-Q M--UP--- 3d-
haskell             ------S- ------L- ---
gquic               VHDCRZ-Q M------- 3d-
lsquic              -------- -------- ---
msquic              VHDCRZSQ MBAUP-L- ---
mvfst               VHDCRZ-Q MBA---L- 3dp
ngtcp2              VHDCRZSQ MBAU--L- 3dp
ngx_quic            VHDC---Q M------- 3--
pandora             -------- -------- ---
picoquic            -------- ------L- ---
quant               -------- ------L- ---
quic-go             VHDCRZSQ M--U---- 3--
quiche              VHDCRZSQ M------- 3--
quicly              --D----- -------- 3--
quinn               -------- -------- ---

aioquicMP           VHDCRZ-Q MBAUP-L- 3dp
```

(b) Interop results of MPAIOQUIC.

Figure 37: Interop results.

By further comparing the two figures with each other, another difference can be identified. For the entries of “akamaiquic”, “haskell”, “lsquic” and “quant” no result flags were given for the MPAIOQUIC implementation, while the AIOQUIC implementation does have certain result flags. The reason for this difference is that, during the implementation period, a number of newer QUIC versions have been released. This meant that the server endpoints were also updated, and no longer accepted connections from out MPAIOQUIC-client that ran an older version of QUIC. Thus, no connection could be established to perform the tests, and therefore no result is shown in the summary.

From the remaining entries in the summary table we can assume that the MPAIOQUIC implementation behaves at least as well as the regular AIOQUIC implementation would when connections are established with remote peers running different implementations that do not support the MultiPath extension.

7.2.2 PQUIC with MultiPath plugin

Besides running interops with the other implementations to evaluate the behaviour of our MPAIOQUIC implementation when regular connections are used, we also tried to perform interops with the PQUIC implementation. Initially, the developers of PQUIC provided a public server endpoint to which we could connect using out MPAIOQUIC-client. However, this approach would also come with a drawback, because we would only be able to collect logs from our client, while the logs from the server remained inaccessible. Additionally, an inverse interop, where a PQUIC-client connects to our MPAIOQUIC-server, would have to be done separately. Thus, a different approach was taken to perform interops between the two implementations. In our approach, both implementations would be able to establish a server and/or client endpoint on a laptop, and communicate via "localhost". See Figure 38 for a visualisation.

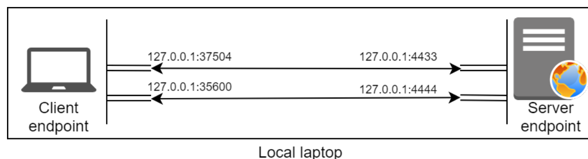


Figure 38: An example interop setup.

Both the MPAIOQUIC implementation and the PQUIC implementation provided functionalities to establish a local client and/or server endpoint. In MPAIOQUIC, the http3_server.py-file and the http3_client.py-file could be used to set up an endpoint that supported the use of MultiPath (and logging via the QLOG-scheme). Certain parameters could also be given upon executing the files, see section 6.2.1 for an explanation. In PQUIC, a picoquicdemo-executable can be generated to set up an endpoint. However, this executable does not directly provide an endpoint that supports the use of MultiPath, or logging via the QLOG-scheme. For both concepts to be supported, additional plugins had to be given as parameters to the executable. Due to lack of documentation, it was not immediately clear which plugin-files had to be included when running the executable. For example, there are six plugin-files that enable the use of MultiPath in a PQUIC endpoint. However, for a PQUIC endpoint to support MP in a connection with an MPAIOQUIC endpoint, only two of these plugin-files could be used, where the exchange of MP-transport parameters, as well as a MP-packet scheduler are supported [32]. Additionally, to enable logging via the QLOG-scheme, two additional plugin-files had to be included [33]. A first plugin-file would enable logging of events that can occur during a regular QUIC connection, while the second plugin-file would enable logging of additional events that can occur during a MultiPath QUIC connection. Lastly, to ensure that the executable only used IP-addresses that only used "localhost", an additional address filtering plugin had to be given as a parameter to the executable. A total of four plugin-files were required to set up a PQUIC endpoint.

After including all the required plugin files, a local PQUIC endpoint could be set up and interop tests could begin. An important note that had to be considered during these tests was that the migration-feature, along with the issuing and retiring of Connection IDs was not yet implemented by PQUIC. From here, initial connection attempts were made between an MPAIOQUIC-client and a PQUIC-server. During these initial connection attempts, the handshake phase completes successfully, but there were some initial problems on the MPAIOQUIC-client regarding the reception of MP_NEW_CONNECTION_ID-frames.

A first problem occurred when the MPAIOQUIC-client included the max_sending_uniflow_id-transport parameter with a value of 0, indicating that it did not want to use additional sending uniflows to send data to the peer. However, the PQUIC-server did not check for the value of this transport parameter, and proceeded with the idea that the client had advertised a value of 1, indicating that it wants to use a second sending uniflow (with id 1). Thus, the PQUIC-server sends a MP_NEW_CONNECTION_ID-frame to communicate a connection ID for uniflow 1 to the client. This was incorrect, because the client did not have a sending uniflow with ID 1, resulting in an error being thrown [34].

A second problem occurred when the MPAIOQUIC-client gave a value greater than 0 to the max_sending_uniflow_id-transport parameter, thus indicating that it wanted to use more than one sending uniflow to send data. In this case, the PQUIC-server also communicated a new connection ID for uniflow 1 to the client via a MP_NEW_CONNECTION_ID-frame. Here, the client was able to "accept" the frame, since it had a sending uniflow 1. However upon pulling data from the frame, incorrect values were read. For example, the frame carried a retire_prior_to-value of 8, and a length value that could vary between 34 and 191. The combination of these two values resulted in the client reading incorrect data and throwing a series of errors, along with incorrect frame type identification. From these errors the MPAIOQUIC-client could not recover and the connection had to be closed, See Figure 39 for an example error.

```
=== pquic test_version_negotiation ===

2020-10-13 10:29:50,570 INFO quic [6b91518f7c95fe4f] Retrying with QuicProtocolVersion.DRAFT_29
2020-10-13 10:29:50,630 INFO quic [6b91518f7c95fe4f] ALPN negotiated protocol h3-29
2020-10-13 10:29:50,631 WARNING quic [6b91518f7c95fe4f] Error: 7, reason: Failed to parse frame, frame_type: 64
```

Figure 39: A frame parse error leading to connection termination.

Our initial guess was that we missed something when decrypting the 1RTT packet that carried the frame, because in the draft of Q. De Coninck et al., a security consideration was noted where the nonce computation of packets had to be done differently. Apparently, the PQUIC implementation also did not implement this changed nonce computation, thus the problem resided elsewhere. Further inspection led to the discovery that the PQUIC implementation did not include all the required fields in the newly defined MP-frames [35]. For example, no retire_prior_to-field was included in the MP_NEW_CONNECTION_ID-frame, resulting in all other field values shifting to the left. This explained why the MPAIOQUIC-client pulled a value of 8 for the retire_prior_to-field, since this was originally the value of the length field. Besides the MP_NEW_CONNECTION_ID-frame, the ADD_ADDRESS-frame also did not include all the required fields, because the interface_type-field was left out.

After the developers at UCLouvain fixed these issues, additional interop tests could be performed between the MPAIOQUIC- and PQUIC implementation. Here, additional discoveries of missing features and issues were made, because the PQUIC implementation also did not yet support the creation and processing of MP_RETIRE_CONNECTION_ID-frames and REMOVE_ADDRESS-frames. Besides these missing features, it was discovered that certain packets were not being decrypted successfully by either endpoint implementation [36, 37]. In Figure 40 a connection between a MPAIOQUIC-client (left) and a PQUIC-server (right) can be seen (generated via the QVIS visualisation tool). This connection shows that three packets are marked as dropped, visualised via a red “X”. (In reality, these packets did arrive at the receiver, but were not logged as received, and caused a decryption error.)

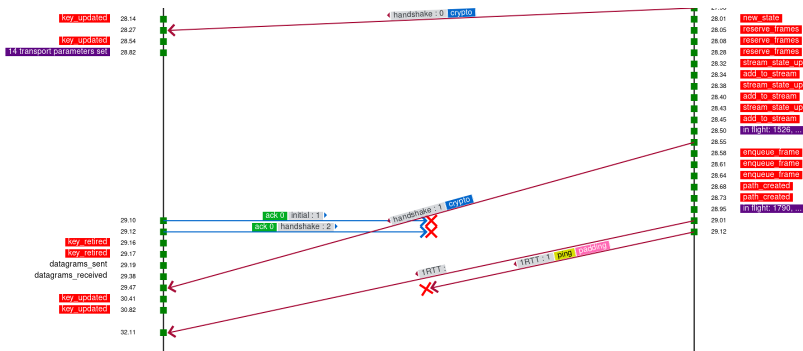


Figure 40: Decryption errors occurring.

This issue was not unique to the MultiPath implementation, as the same decryption issues were also identified in connections between the regular AIOQUIC and PQUIC implementations on the public interop runner. The decryption issue in AIOQUIC for the 1RTT packet (the third “X” on the MPAIOQUIC client) is the result of the packet having a size that is larger than what the AIOQUIC decryption mechanism allows (a maximum of 1500 bytes was allowed). Thus, the AIOQUIC implementation receives the packet, but does not log it, nor can it successfully decrypt it, and therefore it can also not acknowledge the packet as received. In a regular connection between AIOQUIC and PQUIC, the PQUIC-server will later on identify that this packet is marked as lost, and continues the connection (and data transmission) normally, without really causing further problems.

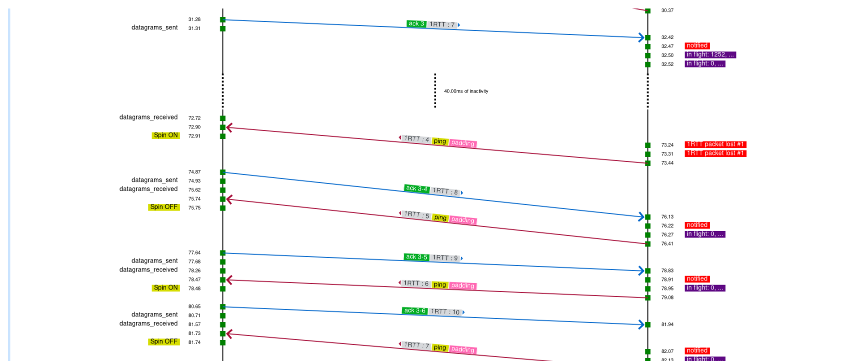


Figure 41: PING-ACK-loop occurring after packet loss.

However, in a MultiPath connection between the MPAIOQUIC-client and the PQUIC-server, the PQUIC-server would also identify that this packet was lost. But instead of continuing the connection normally, from this discovery the server now resorted to continuously sending PING-frames to the AIOQUIC-client, for which the client can acknowledge the packets and respond correctly. This PING-ACK-loop is repeated several times, after which the connection is terminated due to no progress being made (according to the PQUIC client). See Figure 41 for a visualisation. We found a quick solution to this problem, where the MPAIOQUIC-client sends an additional max_udp_payload_size-transport parameter with a value of 1440 to the PQUIC-server. This causes the server to reduce the size of the packets that it sends, and thus the aforementioned 1RTT packet is now successfully decrypted and acknowledged by the client. This fix did not lead to a connection being completed successfully however, because the PQUIC-server stopped sending packets to the MPAIOQUIC-client after a while. See Figure 42 for a visualisation.

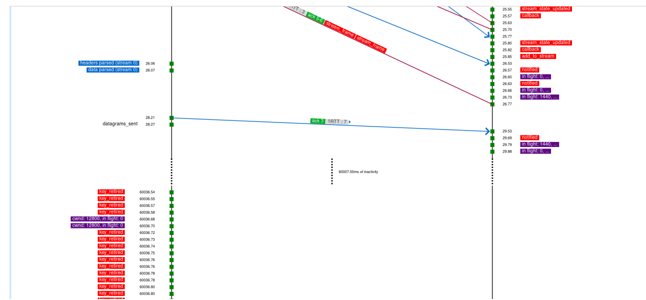


Figure 42: PQUIC server stops sending.

Besides these tests between an MPAIOQUIC-client and a PQUIC-server, an additional test was performed where the roles of client and server were inverted. In this test, the MPAIOQUIC-server was able to establish a connection with a PQUIC-client, where the server also was able to establish two additional sending uniflows towards the PQUIC-client via the path validation procedure. However, when the server started to transmit stream data over the multiple sending uniflows, the PQUIC-client suddenly stopped responding to additional packets. This was because the PQUIC-client crashed upon trying to generate MP_ACK-frames, and was unable to close the connection with the server. Additionally, the PQUIC-client also generated an incomplete QLOG-files. See Figure 43 for a visualisation.

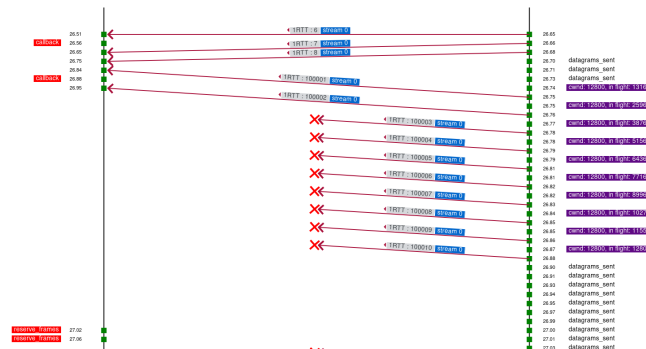


Figure 43: PQUIC client crashing when generating MP_ACK-frame.

From the bug reports, the developers at UCLouvain identified an issue in the PQUIC implementation regarding the usage of the initial uniflows. An initial fix for the PING-ACK-loop and the PQUIC-server halting the sending of packets (due to a crash) was delivered later on. With the fix, the PQUIC implementation now uses and converts initial paths into uniflows instead of opening new (redundant) uniflows. The developers also noted that, in their testing, this has simplified things quite a bit and fixed some additional bugs.

Further testing between the MPAIOQUIC implementation and the fixed PQUIC implementation led to the discovery of new issues being introduced, when the endpoints try to establish additional sending uniflows. When using an MPAIOQUIC-server and a PQUIC-client, the server tries to validate additional paths to enable the use of additional sending uniflows. However, the PQUIC-client was unable to respond in a correct manner. The PQUIC-client did indeed receive the PATH_CHALLENGE-frames from the server, but upon sending PATH_RESPONSE-frames the packet that carries these frames never arrives on the MPAIOQUIC-server. This is because the PQUIC-client also tries to validate "a" path on its own, while sending this packet. See Figure 44 for a visualisation. Additionally, the PQUIC-client sends UNIFLOWS-frames at incorrect times, where it sees uniflows as active before the path has been validated.

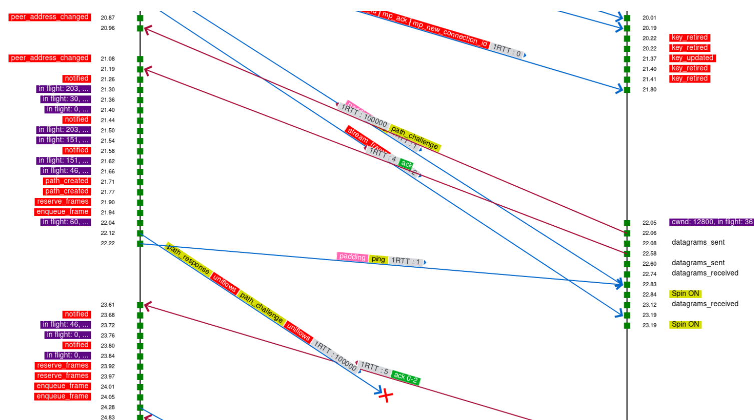


Figure 44: PQUIC client incorrectly trying to validate a path.

Besides this, the connection is able to continue without additional uniflows, and transmission of stream data can be completed in full. However, the PQUIC-client never sends an ACK-of-ACK-trigger to the MPAIOQUIC-server, and starts to identify almost all packets as lost. At certain points during the connection, the PQUIC-client decides to send additional PATH_CHALLENGE-frames to the MPAIOQUIC-server, which are successfully received and acknowledged by the server. These packets carrying the frames also trigger an ACK from the server, where all the previously received packets are acknowledged. (Thus, the packets are not really lost as the PQUIC-client thinks, instead the server is just waiting for an ACK-trigger to acknowledge all the packets at once, which is correct behaviour). Lastly, after the sever has completed the stream data transmission, the PQUIC-client still tries to perform multiple path validation procedures before it realizes that it has received all the stream data. At this point the PQUIC-client halts the procedures and closes the connection.

In the inverse case, a successful connection between a MPAIOQUIC-client and a PQUIC server is performed, where all the requested data is exchanged. In this connection, the MPAIOQUIC-client is able to establish an additional sending uniflow via the path validation mechanism. Additionally, the client is also able to send acknowledgements to the PQUIC-server, and the PQUIC-server is able to send back MP_ACK-frames to acknowledge the packets that the client sent over sending uniflow 1. Lastly, the client is also able to process these MP_ACK-frames correctly. In this connection, it must be noted that the PQUIC-server never attempted to validate an additional path, nor did it try to use an additional sending uniflow to send data to the client.

Based on our findings, the developers at UCLouvain discovered an additional bug that caused memory corruption, and therefore the mentioned packets from the PQUIC-client were not being sent to the correct address. A new update was pushed that tried to fix this issue, however the mechanisms that provided logging via the QLOG-scheme failed to log correctly. Thus, no further inspections and tests could be performed. At the time of finishing this thesis, a new fix was pushed to correctly log the events that occur during a connection. However, no additional tests have been performed to check if the connection performs as expected, where multiple uniflows can be used by both endpoints to transport data.

7.3 Evaluation results

By performing two separate interops, one with all the other implementations on regular QUIC connections, and the other one with the PQUIC implementation on MultiPath connections, a series of findings have been found. First, the interops with all other implementations showed that the MPAIOQUIC implementation behaves correctly when the other implementations do not support MultiPath. No additional sending uniflows are being established, no MultiPath-based frames are exchanged. The MPAIOQUIC implementation gives the same results as the regular AIOQUIC implementation.

Second, the interops with the PQUIC implementation were a slow and time-consuming process. As of finishing this thesis, the interop tests are still unsuccessful, and connections still have some issues. However, via these interop tests a series of issues in the PQUIC implementation were identified and, in cooperation with the other developers at UCLouvain, fixed. The following list sums up the identified issues.

1. Missing documentation for the use of the MultiPath plugin is clarified;
2. Missing documentation for the use of the QLOG plugin is identified;
3. Highlighted the ability to filter IP-addresses based on certain preferences;
4. The max_sending_uniflow_id-transport parameter is now taken into consideration upon connection establishment;
5. Incomplete frames (MP_NEW_CONNECTION_ID and ADD_ADDRESS) now contain all required fields;
6. The decryption issue of 1RTT packets can be fixed by including a max_udp_payload_size-transport parameter;

7. Lost 1RTT packets are now correctly being handled (instead of PING-ACK-looping);
8. MP_ACK-frames are now correctly generated without crashing the endpoint;
9. Additional attempts at establishing new sending uniflows are performed;
10. Correct use of the QLOG logging scheme is implemented;
11. Identified the incorrect use of UNIFLOWS-frames;
12. Identified the fact that no ACK-of-ACK triggers are sent;

8 Discussion and Future work

While the concept of MultiPath was planned to be added in a later version of QUIC, multiple developers already started implementing the concept as an extension for the current version of QUIC. As mentioned in section 5, there already are three draft proposals active, each of them taking a different approach on how the concept of using multiple paths should be implemented. In general, the concept can be broken down in three big parts, each containing a set of mechanisms to allow the MultiPath operation to function. The first part is a path manager that is responsible for the establishment and removal of additional paths in a connection. This part is already discussed in section 5. A second part is the MultiPath congestion controller that has the responsibility of achieving fairness towards other connections in a bottleneck link, while also redirecting traffic away from congested areas in the network. This part is discussed in section 8.1. The last part is a packet manager that has the responsibility of deciding how packets should be transported over the multiple available paths. This last part is discussed in section 8.2.

The combination of these mentioned parts allow a connection to use multiple paths simultaneously. However, by having multiple developers approach the concept differently, and design and evaluate proposed mechanisms for each part, a series of discussions have arisen about the implementation and usefulness of MultiPath in QUIC. A discussion about this is given in section 8.3. Lastly, the MPAIOQUIC implementation prioritized establishing and performing interoperability tests with the PQUIC implementation. These tests were mainly focused on the path manager mechanisms from the draft of Q. De Coninck et al., leaving less time on the evaluation of the other two parts mentioned previously. A discussion about the implementation of these parts, and additional missing features can be found in section 8.4.

8.1 Implementing MP Congestion Controller(s)

Whenever an endpoint wants to transport data to its peer, it should send the data in such a way that it does not overload the reception capabilities of its peer. Additionally, the sender also has to ensure that it does not overload the transport capabilities of the intermediate links over which the packets are transported. To do this, two mechanisms are used, the first one being flow control, explained in section 8.1.1. The second one being congestion control, explained in section 8.1.2.

8.1.1 Flow control

A receiving endpoint may not be able to receive data at the same rate that the sender is transmitting. This can be caused by multiple factors, such as the receiver having older hardware, or the receiver not wanting to reserve many resources for the connection. This results in an upper bound of data transmission being defined that the sender must not cross, risking data to be lost if ignored. In regular TCP, this upper bound is supported by allowing the receiver to send receive window values to the sender. Such a receive window value is present in every packet from the receiver and indicates how many bytes the receive buffer supports for data that has not yet been processed. Any additional bytes transmitted by the sender will be dropped by the receiver.

A sender is, at maximum, allowed to send the indicated amount of bytes to the receiver before it has to wait for the receiver to acknowledge the data. When the receiver has processed the data it will send back an acknowledgement value that indicates how many consecutive bytes have been processed successfully. Thus, for the sender, this acknowledgement value indicates the starting location of a window in which it is allowed to transmit data. In combination with the receive window value, the sender is able to determine the ending position for the current window, See Figure 45 for a visualisation. Every time an increased acknowledgement value is seen by the sender, this window can be shifted further and further over the stream of data that needs to be transported.

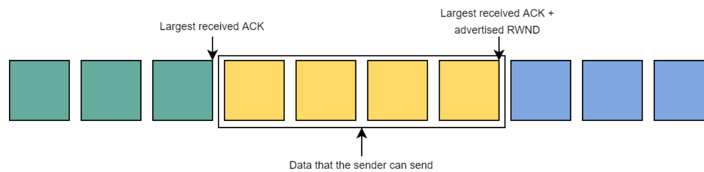


Figure 45: The window in which the sender can transmit data.

In regular QUIC, the support for this upper bound is approached differently. QUIC is designed for endpoints to communicate over multiple streams, possibly even in both directions. The stream data is carried in STREAM-frames that indicate the exact location where the data needs to be placed by a receiving endpoint in an indicated stream. This design forces an extended approach, because unlike TCP, a single receive window will not be sufficient. First, a receiver advertises the limit of total bytes it is prepared to receive for the entire connection, across all streams. This is done via the initial_max_data-transport parameter. The limit value is conceptually similar to the receive window values sent by the receiver in a TCP connection.

Second, a receiver indicates a maximum number of different streams that can be initiated by the peer. This is done via the initial_max_streams-transport parameters, and prevents a sender from concurrently opening too many streams to send data, therefore limiting the amount of resources that have to be reserved by the receiver. Lastly, a receiver also advertises the limit of total bytes it is prepared to receive on a given stream. This is done via the initial_max_stream_data-transport parameters, and prevents a single stream from hogging the connection's limit when multiple streams are active. These limits tell a sender how much data can be sent over a connection, via a given limited set of streams, where each stream is also limited in the amount of data that can be sent.

Unlike TCP, the reception of an ACK-frame does not shift the window in which a sender can send data. Instead, an ACK-frame only indicates that certain packets have been successfully received and processed by the receiver. Therefore the data, that was placed in STREAM-frames carried by these acknowledged packets, no longer needs to be (re)transported to the receiver. Every time a STREAM-frame is sent via a 1RTT packet the number of bytes are added to a total that is tracked by the sender. If the total equals the maximum value for the given stream, the sender is no longer allowed to send additional data over that specific stream, even if more data is still to be sent. Additionally, each of these stream-specific totals are also added together and compared to the connection limit value.

If the sum of the streams' total values is equal to the connection's max value, the sender is no longer allowed to send any stream data at all. To allow the sender to continue sending the remainder of the data, the receiver must provide its peer with increased limits to either a specific stream, the entire connection, or both. This is done by sending MAX_STREAM_DATA and MAX_DATA-frames that indicate larger limits for a given stream and the entire connection respectively. This mechanism introduces two levels of flow control: the stream-level, which prevents a single stream from consuming the entire receive buffer, and the connection-level, which prevents senders from exceeding a receiver's buffer capacity for the connection.

8.1.2 Congestion control

While the flow control mechanisms in TCP and QUIC ensure that the sender does not overwhelm the receiver, an additional consideration must be taken about the transport capabilities of intermediate links in the network. These links can differ greatly from one another, and therefore their transport capabilities are different as well. Additionally, data from many different connections (not just TCP or QUIC connections), often passes through the same link in a network, resulting in each of these connections having to share the capabilities of the link. If only a flow control mechanism is used, senders would transmit all the data that fits inside the advertised limits, preferably as fast as possible.

If multiple senders do this, a large amount of packets would arrive at intermediate links. Each intermediate link must process each of these incoming packets to some degree to propagate them to the next link that is closer to the destination. A buffer is provided to allow multiple packets to be queued. However, with no additional control, this buffer is quickly filled up, and additional packets will have to be dropped due to lack of buffer space. This results in congestion where the data from the many different connections that pass through the link will not arrive at the destinations. To prevent congestion from happening, an additional mechanism must be introduced that prevents a sender from overloading the links in the network.

In regular TCP, a congestion controller is introduced that limits the rate at which data is transported by the sender. During the lifetime of a connection the controller keeps track of a congestion window, whose value is never communicated to the receiver, unlike the receive window. The size of this congestion window is manipulated by the controller via an additive increase/multiplicative decrease scheme that adapts to the evolving state of the network over which the packets are sent. For every sequentially increasing ACK value that the sender receives, the congestion window size is increased (additive increase). This then allows the sender to send more data over the network in accordance to the window, up to the advertised receive window of the receiver.

Whilst this is happening, the sender also measures the round-trip times (RTTs) of each packet that is sent, and takes an average value (and a variance) from a series of packets. An RTT is the amount of time that it takes for a packet to be sent to the receiver, and the sender receiving an acknowledgement from the receiver that confirms the reception of the packet. In Figure 46 the example RTT of a packet containing “Hello” is 74ms.

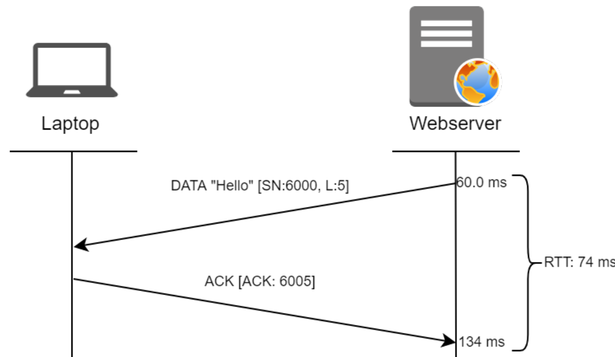


Figure 46: A packet's RTT of 74 ms.

While the congestion window grows over time, it is possible that the amount of data that is being transported by the sender surpasses the transmission capabilities of a link on the network. This will cause the packets to be delayed, or even dropped by the link. At this point, a number of scenarios can occur, such as packets arriving delayed at the receiver, no packets arriving at the receiver, or only some packets arriving at the receiver. Either way, the congestion controller can detect the congestion on the network via two methods. The first method indicates congestion when the sender does not receive an acknowledgement from the receiver before a given timer runs out. This timer is based on the average RTT of the connection (and the variance), and starts when a packet is transmitted by the sender. The second method revolves around the sender receiving a duplicate acknowledgement, which is transmitted by the receiver when it has processed a packet with sequence numbers that are larger than expected.

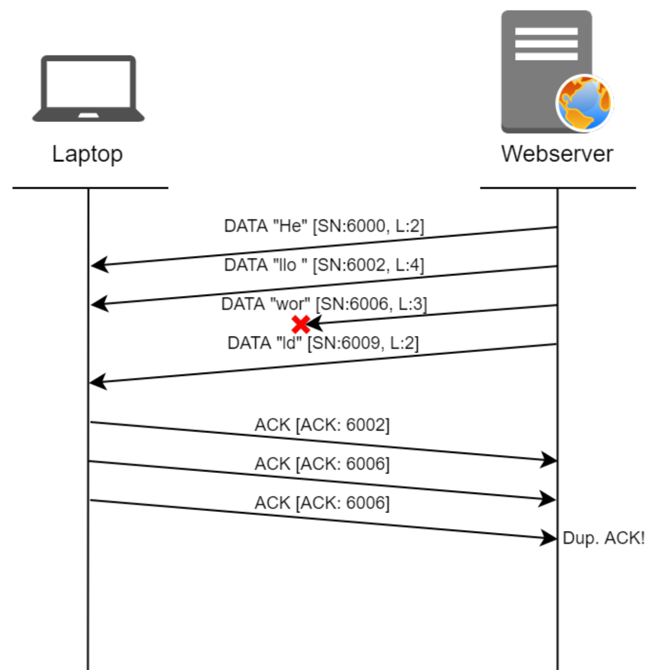


Figure 47: A duplicate ACK occurring.

For example, in Figure 47 the third packet is lost, but packets 1, 2 and 4 do arrive at the laptop. The laptop will detect that packet 4 has an out of order sequence number, and therefore sends a duplicate acknowledgement to the sender. If the congestion controller detects congestion via either method, it will halve the size of the congestion window (multiplicative decrease), restricting the sender in reducing the amount of bytes that it can transmit. This will result in fewer packets arriving at the intermediate link, allowing it to keep accepting new incoming packets. Additionally, the sender will also have to start a retransmission from the first packet that was marked as lost, providing each packet with the same sequence number and data as before. From this point, the congestion controller will return to increasing the congestion window upon the reception of a sequentially increasing ACK value from the receiver. If another congestion event is detected, the window will be halved again. This process is repeated continuously for the remainder of the connection.

For QUIC, this process of increasing and decreasing a congestion window is similar, with some key differences being introduced to the protocol [38]. A first difference is the ability of packets to carry frames that are independent of the packet that carries them. This allows QUIC to use a monotonically increasing packet number in which each packet number is sent at most a single time. Additionally, a receiving endpoint is able to acknowledge these packets via an ACK-frame that carries these packet numbers. Instead of just acknowledging a single packet, multiple ranges of packet numbers are included in the ACK-frame. If a certain packet with a given packet number is lost, the range of acknowledged packets will be split up and a gap will be introduced to indicate the loss. For example, in Figure 48 the third packet is lost, but packets 1, 2 and 4 do arrive at the laptop. The laptop will send an ACK-frame back to the webserver in a 1RTT packet, and the frame will contain two ranges of acknowledged packets. The first range acknowledges packets 1 and 2, and the second range acknowledges packet 4.

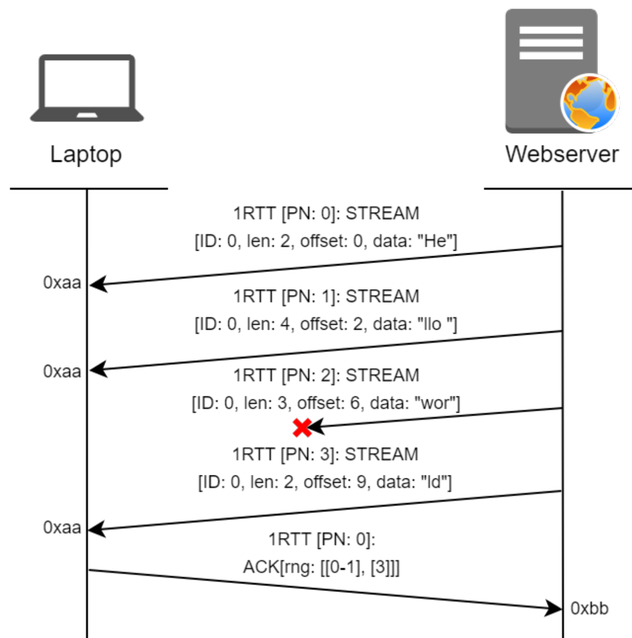


Figure 48: An ACK gap occurring.

The webserver will then detect a timeout of the third packet via two possible methods, the first one being a trigger that signals loss if enough packets that were sent later have been acknowledged. The second method uses a timeout mechanism that triggers after a certain amount of time when packets that were sent later have been acknowledged. If such a packet is declared as lost, the frames that were placed in that packet are then placed in a new packet that carries a higher packet number. This removes disambiguation between transmissions and retransmissions of packets, and allows for certain mechanism to be simplified, or even to be improved. For example, a Fast-Retransmit phase in the congestion controller can be applied universally, based only on a packet number.

A second difference is the fact that QUIC allows a sending endpoint to better measure the RTT value for a given path. In comparison to TCP, a receiving endpoint measures the delay between when a packet is received and when the corresponding acknowledgment is sent. This measurement is then placed inside the ACK Delay-field of an ACK-frame and transmitted to the sending endpoint. When the sending endpoint measures and calculates the RTT value estimates and variances, this communicated delay is taken into consideration, allowing for better estimations and variances. A third difference is that, unlike TCP, QUIC allows for a connection to migrate to a different 4-tuple during its lifetime. If this action is taken by either endpoint, the congestion controllers and RTT-estimators must be reset to default values. This is because the capacity available on the new path might not be the same as on the old path, and if no reset is performed an endpoint could transmit too aggressively until its congestion controller and RTT-estimator have adapted, going against the principle of achieving fairness.

8.1.3 Achieving fairness

The congestion controllers in TCP were designed in such a way to ensure that they do not (continuously) overload the transport capabilities of the intermediate links. From the perspective of the congestion controller, the transmission capabilities of a path are limited by the capabilities of a bottleneck link in the chain of links that has the lowest available transmission rate. Via the additive increase/multiplicative decrease scheme a connection can obtain this available transmission rate. However, the idea is that multiple connections could pass through this bottleneck link to transport data, therefore each of these connections must somehow "work together" without explicitly communicating to prevent the combination of all data transmissions to overflow that bottleneck's transmission capabilities. This is where the concept of fairness is introduced: if N connections pass through a certain bottleneck link with a given maximum transmission rate R, then each connection should obtain a maximum transmission rate of R/N, allowing all connections to have an equal and fair share of the available transmission capabilities of the bottleneck link.

From a scenario's perspective: A phone, a laptop and a desktop each have established connections to different remote peers. Each connection passes though the same Bottleneck link that has a transmission rate of 90 Mbps. See Figure 49 for a visualisation. If all three connections play fair with each other, they will each try to take a third of the transmission capabilities that the bottleneck link has to offer. Thus, each connection will reach a transmission rate of 30 Mbps. If any of these connections further increases its transmission rate, the packets that arrive at the bottleneck link will not fit in the link's receive buffer and therefore have to be dropped, resulting in all three connections being affected.

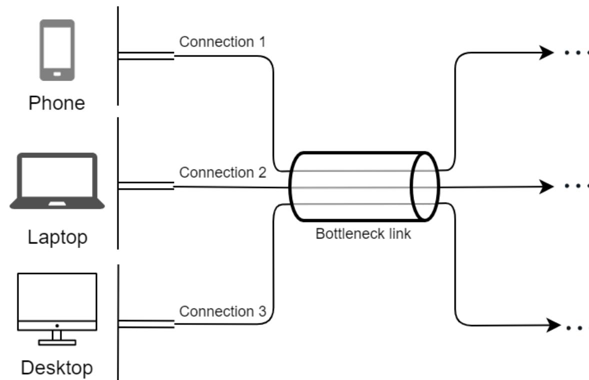


Figure 49: Multiple connections sharing the resources of a bottleneck link.

By providing an additive increase/multiplicative decrease-scheme to the congestion controllers, each TCP connection is able to probe the transmission capabilities of its path, and adapt to changes in the network when, for example, a different connection disappears from the bottleneck link. When congestion occurs, the congestion controller of each connection will reduce its congestion window in a multiplicative manner, where large windows will receive bigger reductions than smaller windows. Combining this with the additive increase, the congestion window of each connection will eventually reach similar sizes, and therefore equal and fair shares of the transmission capabilities of a bottleneck link can be taken. Due to TCP having a widespread use, new protocols such as QUIC must also take this concept of fairness into consideration to prevent the network from being overloaded.

8.1.4 Adding MultiPath to the mix

By including a mechanism such as advertising a receive window in TCP, or the provision of stream-level and connection-level limits in QUIC, a connection can be established between two endpoints where neither endpoint shall overload its peer with data. Additionally, the introduction of a congestion controller allows for a sender to probe and utilize the data transmission capabilities of a given path that the connection currently uses. If the sender detects that a potential congestion event is occurring on the path, it will reduce the amount of data that is being sent to allow the path to recover. For the single path design of TCP and QUIC, the use of a congestion controller allows multiple connections to pass through the same links that are available on the network. Each connection that passes through a certain link will try to share the available resources of that link equally among themselves, without communicating with each other. However, when the idea of MultiPath is added to their design, some considerations around the use of these two mechanisms have to be taken for both of these protocols.

To start, a connection between two endpoints can exist that utilizes multiple paths to transport data. Without considering the transport capabilities of each path, the data that arrives at the receiver still has to be processed before it can be given to a receiving application. This means that data arriving from different paths eventually are placed in the same receive buffer of an endpoint. Therefore, each path will share the same receive buffer space.

For MultiPath TCP, this means that acknowledgement packets sent over each subflow will carry the same value for the advertised receive window. It is possible that the values for these receive windows are altered by middleboxes, resulting in different values arriving at the sender. In this case, the sender only uses the largest recently advertised receive window value as the actual receive window for the connection. For uniflows that have a smaller window the sender only transmits data that fits in the uniflow's receive window. In MultiPath QUIC, the same connection-level and stream-level limits from regular QUIC are used to signal how much data the receiver is willing to accept. In contrast to MPTCP, these limits are communicated via frames that are carried by encrypted packets, and therefore no intermediate node can alter the values.

In comparison, the use of a congestion controller in MultiPath requires more changes to be made from the regular design of both protocols. From a very basic standpoint, a congestion controller manipulates a congestion window to adapt the transmission rate of a path to the changing capabilities of that path. Each path has its own characteristics, and its own capabilities to transport data. This means that it would be an illogical choice to use a single congestion controller with the same congestion window for all the paths, because some paths may not support the transmission rate of others. Therefore, an initial solution could be to apply the idea of a regular connection, transporting data within a congestion window over a single path, multiple times. In this solution, a sending endpoint allocates a separate congestion controller for each individual subflow (in MPTCP) or sending uniflow (in MPQUIC). Each of these congestion controllers would then manipulate a congestion window for a specific path, allowing the sender to adapt the sending rate for each subflow or uniflow separately. This initially ensures that a sender does not overload the transport capabilities of the intermediate links. However, this solution is sub-optimal.

Conceptually, as stated by the idea, by allocating a separate congestion controller to each uniflow, a separate "connection" is created to transport data to a peer, where the congestion controllers from each path do not talk with each other. With this, certain problems can arise due to the MultiPath connection being able to be more aggressive than the regular connections competing with it. A first problem that can be identified is the fact that a MultiPath connection does not play fair when it comes to sharing resources of a bottleneck link that is traversed by more than one of the connection's subflows or uniflows. A second problem comes in the form of resource pooling.

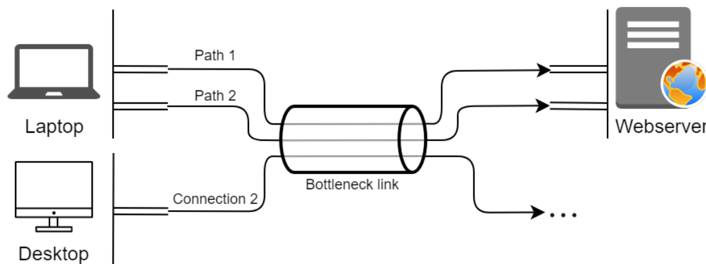


Figure 50: Two connections sharing the resources of a bottleneck.

Observing the first problem, from a scenario's perspective: a laptop has established a MultiPath connection to a remote webserver. In this connection there are two paths that are used simultaneously by the laptop to send data to the webserver, being Path 1 and 2. Both of these paths traverse through a bottleneck link where a different, regular connection also uses the link to transport data, being Connection 2. This other connection comes from a different endpoint, a desktop located somewhere else in the world. See Figure 50 for a visualisation.

Assuming that all the congestion controllers from both paths on the laptop and from the second connection on the desktop play fair in general, they will each try to obtain and use a third of the transmission capabilities that the bottleneck link has to offer. Thus, a third is taken by the first path, another third is taken by the second path and the last third is taken by the second connection. This results in the connection between the laptop and the webserver taking two thirds from the available transmission capabilities of the bottleneck link, while the second connection only receives a third. From the perspective of the regular connection, this is an unfair situation, because the MultiPath connection takes more than its fair share of the bottleneck link's transmission capabilities. In comparison, an ideal and fair situation for both connections can be established when each connection only takes half of the link's transmission capabilities.

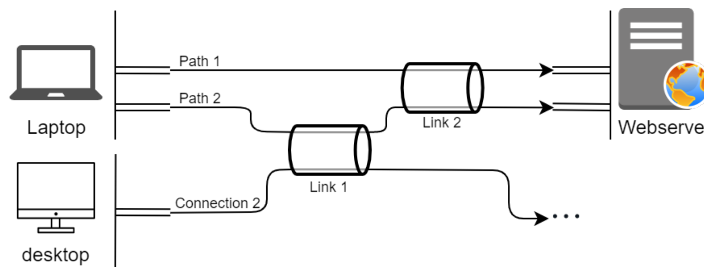


Figure 51: Two connections pooling the resources of a network.

Now observing the second problem from a scenario's perspective: a laptop has established a MultiPath connection to a remote webserver. In this connection there are two paths that are used simultaneously by the laptop to send data to the webserver, being Path 1 and 2. There also exists a second, regular connection that comes from a desktop located somewhere else in the world, being Connection 2. Path 2 and Connection 2 first pass through Link 1, after which both paths from the MultiPath connection pass through Link 2. See Figure 51 for a visualisation. In this scenario both links have the same transmission capabilities, being 100 Mbps.

Assuming that all the congestion controllers from both paths on the laptop and from the second connection on the desktop play fair in general, they will each try to obtain their fair share of transmission capabilities on the links. The congestion controllers from Path 2 and Connection 2 will both take half of the transmission capabilities of Link 1, and the congestion controllers from Path 1 and 2 will also take half of the transmission capabilities of Link 2. This results in both paths and the additional connection having a data transmission rate of approximately 50 Mbps. Additionally, via both paths the laptop gains a total transmission rate of 100 Mbps for its MultiPath connection, whereas the desktop would only receive a total transmission rate of 50 Mbps. From a fairness perspective this situation would be optimal for each link, since the transmission capabilities are shared equally.

However, from a resource pooling perspective the combined use of both links is not optimal. In the figure it can be seen that both the laptop and the desktop are competing for the transmission capabilities of Link 1, creating a congested area where packets from both Path 2 and Connection 2 will be dropped if their respective congestion windows grow too big in size. In comparison, there is no competition for Link 2, since only the laptop makes use of it. In this case, a more ideal situation could be established if the laptop reduced the amount of data that is being transmitted over Path 2, and move the traffic over to Path 1. This reduces the pressure on Link 2, and allows the other connection from the desktop to claim more and more of the transmission capabilities of Link 1. If this were to happen, the transmission rate for the MultiPath connection between the laptop and the webserver would stay the same, while the transmission rate for the connection from the desktop would increase, up to a maximum of 100 Mbps.

To obtain these ideal situations in both scenarios, a better solution must be introduced to the congestion controllers in a MultiPath connection. It is a good start to include a congestion controller for each path individually, so that the transmission rate on a path can be adjusted according to the changes that occur on the path. However, an additional step can be taken where the congestion controllers can communicate with each other. Since each congestion controller manipulates its own congestion window via an additive increase/multiplicative decrease scheme, a solution could be introduced in the form of an algorithm that links the window-increase functions of each congestion controller. In [7], three goals are given to this algorithm to ensure that (i) it provides the congestion controllers with fairness at bottleneck links, and (ii) the concept of resource pooling is applied, where traffic is moved away from congested areas in the network. These goals are (1) to improve throughput, (2) to not harm other connections, and (3) to balance congestion. To achieve these goals via an algorithm that links the increase functions of a congestion window together, certain changes have to be introduced. As an example algorithm, OLIA [8] will be used.

In a regular connection, the congestion controller increases its congestion window with an increase-value that is calculated via the amount of bytes that have been acknowledged by the peer. For a MultiPath connection, this calculated increase-value is not added in full to the congestion window of a path. Instead, a modified increase-value is added. The modification comes in the form of multiplying the increase-value with a decimal representation of the share that the path's congestion window has in relation to the total sum of all congestion windows in the connection. A simplified version of the first term in OLIA's additive increase algorithm can be seen in equation (1). This modification is applied in such a way that, if each path adds an increase-value to their congestion window upon receiving an acknowledgement for that specific path, the summation of these values would equal that of the increase-value a regular connection would get in the same scenario. This is to ensure that multiple paths belonging to the same connection do not take more than their fair share of the available transmission capabilities on a given bottleneck link, therefore accomplishing the second goal.

$$CWND_i \mathrel{+}= increase * \frac{CWND_i}{(\sum_{p}^{paths} CWND_p)^2} \tag{1}$$

Applying this to the first scenario denoted in Figure 50 at a certain moment in time, Connection 2 would increase its congestion window with an increase-value of X, and Path 1 and 2 would increase their individual congestion windows with a modified increase-value of Y and Z. Assuming that they all have sent the same amount of data to their respective receiver, and assuming that they all have an equal sized congestion window, then the combined increase-value of Y + Z should be equal or close to the increase-value of X.

This modification only ensures that the congestion window of each path is increased in a fair manner. There currently is no parameter that allows an endpoint to redirect traffic to less congested paths. OLIA introduces an additional term to the modification to make redirection possible. The term (ALPLHA) is added to the decimal representation of the share that the path's congestion window has, and its value is determined based on how the current path ranks among the others in the connection. To rank the paths, three different sets are introduced, being best_paths, max_w_paths and collected_paths. Each of these sets can contain a number of paths that currently meet a certain requirement.

The set of best_paths contain the paths that have the best "average number of sent bytes between losses"-to-RTT-ratio. These paths are considered the best to use for the connection at that moment, since they have a good average RTT and experience low to no congestion events. Additionally, the set of max_w_paths contains the paths of the connection that currently have the largest congestion windows, indicating that they currently can transport the most data. Lastly, the set of collected_paths contains the paths that are in best_paths, but not in max_w_paths, meaning that these paths currently have potential to redirect traffic towards them, since their loss rates are low, have a good RTT, and have a smaller congestion window that should be increased. Note that not every path must be listed in these sets, these unlisted paths are in general not the best paths available to the connection, and therefore should not be prioritized at the current moment. A simplified version of the first and second terms in OLIA's additive increase algorithm can be seen in equation (2).

$$CWND_i \mathrel{+}= increase * \left(\frac{CWND_i}{\left(\sum_{p}^{paths} CWND_p\right)^2} + \frac{ALPHA}{CWND_i}\right) \tag{2}$$

The value of ALPHA is determined based on which set the path currently belongs to. If the path is part of the collected_paths-set, the value of ALPHA is calculated between 0 and 1 to allow faster increase of the congestion window for that path. If the path is part of the max_w_paths-set, and there currently are other entries in the collected_paths-set, the value of ALPHA is calculated between -1 and 0. This indicates that the congestion window of the path should grow in smaller increments, or even reduce in size. In all other cases, the value of ALPHA is 0.

By continuously updating these sets, and updating the congestion windows of each path in accordance to a calculated ALPHA value, a mechanism is introduced where the value of ALPHA changes continuously, and that allows for potential better paths to grow their congestion window, while the growth of congestion windows for the currently used paths stagnates, or even inverts. This, over multiple round-trips, leads to traffic being redirected to less congested areas in the network, and therefore accomplishes the third goal.

Applying this to the second scenario denoted in Figure 51 over a longer time-span, assuming that the RTT of Path 1 is a little bit higher than the RTT of Path 2 (resulting in a slower congestion window increase). Both the congestion windows of Path 2 and connection 2 will reach a certain size that results in both having a fair share of the transmission capabilities of Link 1. However, upon nearing the maximal transmission rate of Link 1, the RTT values and the number of congestion events will increase on Path 2. This will result in the path only being placed in the max_w_paths-set, while Path 1 is placed in both the best_paths-set and the collected_paths-set. At this point, OLIA will allocate a positive ALPHA-value to the increase of the congestion window of Path 1, whereas a negative ALPHA-value will be allocated to the increase of the congestion window of Path 2. Over time, this will lead to the congestion window of Path 1 growing more and more, while the congestion window of Path 2 will stagnate or even shrinks. Eventually, this will lead to traffic being redirected towards Path 1.

Lastly, by prioritising the increase of the congestion windows of the best paths in a connection, OLIA will get a total rate that is at least as good as the rate a normal connection would get on the best path available to it, therefore accomplishing the first goal. It is important to note that the equations used here are a simplified version of the actual equations used in OLIA. The actual equations also include the RTT-values of each path to account for differences in RTT between the paths.

In general, a MultiPath congestion controller aims to accomplish the three mentioned goals to obtain an increase in throughput and robustness to link failure. As shown in the two scenario's, these congestion controllers must try to remain friendly to other regular connections, and stay responsive to changing events in the network. However, there's an inevitable trade-off between friendliness and responsiveness that was mathematically proven in [9]. In the same paper, the argument is given that OLIA remains unresponsive to changes that happen on the network. Taking an example from scenario 2, if Connection 2 would disappear from Link 1 while the laptop primarily uses Path 1, the decimal representation of the share that PATH 2's congestion window has is very small in relation to the total sum of all congestion windows in the connection. Therefore the final increase-value for the congestion window of Path 2 remains very small, resulting in many RTTs that need to happen before the congestion window returns to a steady size (and larger representation).

Overall, there are also other designs that are responsive to changes within the network, but are not friendly towards other regular connections. As an example algorithm, LIA [7] will be used. In comparison to OLIA, the increase-value is modified differently. Here, a minimum is taken from two options, the first one being the increase-value that a regular connection would receive, and the second one being an increase value that is multiplied by an ALPHA value and divided by the total sum of all the congestion windows in the connection. A simplified version of LIA's additive increase algorithm can be seen in equation (3).

$$CWND_i \mathrel{+}= MIN(\frac{increase}{CWND_i}, \frac{ALPHA * increase}{\sum_{p}^{paths} CWND_p}) \tag{3}$$

This modification is applied in such a way where LIA can guarantee that a path cannot be more aggressive than a regular connection in the same situation, therefore achieving the second goal. The ALPHA value is introduced in the second option to manipulate the aggressiveness of a given path. The value is chosen such that the combined throughput of the paths is equal to the throughput a regular connection would get if it ran on the best path, therefore accomplishing the first goal. A simplified version of LIA's ALPHA calculation can be seen in equation (4).

$$ALPHA = \sum_{p}^{paths} CWND_p * \frac{MAX(CWND_i)}{(\sum_{p}^{paths} CWND_p)^2} \quad (4)$$

Notice that there is no second term that is being added to the second option, as LIA does not have a mechanism to redirect traffic to less congested paths. Applying LIA to the first scenario denoted in Figure 50, the same result is obtained as with OLIA. Connection 2 would increase its congestion window with an increase-value of X, and Path 1 and 2 would increase their individual congestion windows with a modified increase-value of Y and Z. Assuming that they all have sent the same amount of data to their respective receiver, and assuming that they all have an equal sized congestion window, then the increase-value of Y + Z should be equal or close to the increase-value of X. However, Applying LIA to the second scenario denoted in Figure 51 gives a different result, because no redirection of data flow is performed. For Path 2, LIA still tries to take its fair share of the transmission capabilities of Link 1, and being unfriendly to Connection 2. This is actually the same situation that occurs in the original issue, the only difference being that the congestion controllers on the laptop link the congestion windows together to prevent the issue that was shown in the first scenario.

However, LIA still remains responsive to changes in the network. For example, if Connection 2 were to disappear from Link 1 in the second scenario, the congestion window of Path 2 would grow faster than in OLIA. Therefore, Both Path 1 and Path 2 would return faster to a steady size. (It is possible that congestion is experienced on both Paths since they share the transmission capabilities of Link 2, but eventually an equilibrium is achieved.)

The differences between the two algorithms in certain situations happen because the algorithms have to choose how the trade-off between friendliness and responsiveness is achieved. Different approaches by other algorithms to the implementation are taken. For example, BALIA [9] is another algorithm that takes yet another approach to the concept, where it tries to explicitly balance the mentioned trade-off. While the three mentioned algorithms (LIA, OLIA and BALIA) are packet-loss-based algorithms, a fourth algorithm called wVegas [10] takes the approach from an entirely different perspective, by operating on a delay-based manner. By letting each algorithm take a different perspective, finding a general approach to MultiPath congestion control is a non-trivial task. However, some general steps can be identified that are taken by (most) algorithm implementations. For example it is generally agreed that the congestion windows of each path have to be linked somehow. Additionally, the increase-values that are added to each congestion window have to be manipulated in some way, to ensure that the paths express fair behaviour to other, regular connections. Lastly, additional parameters should be introduced and manipulated to enable the congestion controller to react to changes in the network, and redirect traffic to less congested paths.

An overview of the mentioned congestion control algorithms can be found in [39]. For the MPAIOQUIC implementation, only a dummy congestion controller is implemented. A discussion about implementing a real MultiPath congestion controller in MPAIOQUIC can be found in section 8.4.1.

8.2 Implementing packet schedulers

By introducing mechanisms such as MultiPath congestion control in combination with receive windows in TCP, or the provision of stream-level and connection-level limits in QUIC, a MultiPath connection can be established between two endpoints where the aggregated bandwidth of the paths allows for more resilience to network failures, while also supporting a larger throughput of data. However, for the connection to make the best use of the increased throughput, there also has to be a mechanism in place in the form of a packet scheduler that can make decisions on how the data should be transmitted over the available paths of a connection.

8.2.1 Why is a packet scheduler required?

In general, a MultiPath connection can be established for different purposes. A sending application might require that the connection establishes a second path to the receiver that is used as a back-up in case of a failure on the first path. Another sending application might want to send its data over multiple paths simultaneously. In the first case, where additional paths are used as backups, the idea of a packet scheduler would be mostly obsolete, since the connection would mostly behave as a single-path connection. However, a packet scheduler comes in handy when at least two different paths are to be used simultaneously.

The reason lies in the fact that applications might have certain goals in mind when transporting data. For example, a sending application might aim for the data to be transported with minimal latency. Additionally, applications, such as streaming applications might often require a minimum bit rate to sustain playback, and therefore aim to obtain a high amount of throughput. Another goal might be to keep the cost of the transmission as low as possible, when for example certain paths use a metered connection. To obtain these goals, it is obvious that the data must be transported over the active paths that the sender has available. However, these paths often have certain constraints tied to them, such as high RTTs or small congestion windows, that constantly change over the lifetime of a connection. Randomly spreading the data over the available paths would be a bad idea.

From the perspective of a scenario: A webserver has a MultiPath TCP connection with a remote laptop. In this connection the webserver can reach the laptop via two paths, being Path 1 and Path 2. The webserver wants to send data (a JavaScript-file) over a single stream in 5 sequential packets to the laptop. Here, it does not apply a packet scheduling mechanism, and sends two packets (marked as 1 and 2) to the laptop via Path 1, which has a low RTT. Then it decides that the third packet (marked as 3) should be sent over Path 2, which has a higher RTT. Finally the webserver decides that the last two packets (marked 4 and 5) should be sent over Path 1. This results in packets 1, 2, 4 and 5 arriving at the laptop in a short amount of time.

The laptop processes them and places the content of packets 1 and 2 in the output stream and sends it to the receiving application. However, upon processing packet 4 and 5 the laptop notices that that the packets do not connect to the last byte on the output stream. Thus, the laptop cannot forward the packets to the receiving application. Finally, after a short delay packet 3 arrives at the laptop, which is then processed and placed in the output stream, along with packets 4 and 5. Lastly, the remainder of the stream is sent to the receiving application. See Figure 52 for a visualisation.

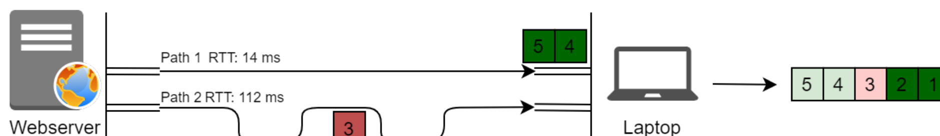


Figure 52: Explicitly sending a packet over a path with high RTT.

In TCP, this situation leads to both the receiving endpoint and the receiving application having to wait for packet 3 to arrive, before the JavaScript code can be handled. At first, the situation seems similar to the TCP HoL-blocking problem, but it is slightly different in the fact that no retransmit has to be performed by the webserver, since the third packet eventually arrives at the laptop. This situation is created due to the webserver not implementing a packet scheduler that should have decided that Path 2 should not be used to send packet 3. The argument could also be made that this problem only occurs for TCP, but that argument would be incorrect. Applying the scenario to QUIC, the laptop could also send the contents of packets 1 and 2 to the receiving application. But then again, the application still has to wait for the contents of packet 3. This is because the JavaScript-file needs to be loaded in its entirety before the JS-code can be handled. Bringing this scenario to a larger scale, where more paths could potentially be used, and each path can have its own characteristics, these types of situations should be avoided to efficiently use the MultiPath connection. Therefore, a packet scheduling mechanism needs to be in place on the sending endpoint. As shown in the scenario, the available paths have certain characteristics that need to be considered. Thus, the packet scheduler must be able to communicate with the congestion controllers that are used on the connection.

8.2.2 MPTCP packet scheduling

There have been multiple proposals for packet schedulers in MPTCP, and each of these have their own approach on how packets should be sent over the different available paths. An initial idea of a packet scheduler can be given in the form of a Round-Robin mechanism. Here, available paths are chosen one after the other in a circular pattern. A packet is sent over a selected path if its congestion window has enough room, otherwise the next path is selected. This initial packet scheduler is part of a larger group of schedulers that do not take into consideration the characteristics of the paths when deciding how a packet should be sent. These schedulers are not really useful in MultiPath scenario's when the path characteristics are heterogeneous, due to the potential performance issues that can be introduced. For example, a Round-Robin scheduler could introduce a similar situation to the one shown in the scenario.

Therefore, a different set of packet schedulers should be used that do take the path characteristics into consideration. These packet schedulers prioritize sending data over the "best" paths on a connection, according to metrics such as average measured RTT. The Lowest-RTT-First mechanism is an example of this set. Here, a set of paths is gathered where the congestion window allows for a packet to be sent over the linked path. From this set, the path with the lowest RTT is selected to send the packet. While on paper this mechanism aims to obtain the goals of a sending application mentioned in the previous section, just sending data on the paths with the lowest RTT is not enough. It is shown in [13] that memory requirements of MultiPath TCP are much higher than the requirements of regular TCP. For devices such as mobile phones that have certain memory constraints tied to them, a smaller receive window than required could be advertised. A MultiPath connection with the lowest-RTT-First scheduler implemented can, in such cases, potentially perform worse than a regular connection would. For example, in the scenario the webserver could receive a small receive window from the laptop. With the Lowest-RTT-First scheduler enabled, the webserver will prioritize sending data over the faster path, in this case Path 1.

However, once the congestion window from Path 1 is filled up while the receive window limit is not yet reached, it will automatically start sending data over Path 2. This is because Path 1 can no longer be used to send additional data, and therefore the path no longer is part of the set of candidates from which the scheduler selects a path. After sending data over Path 2, a certain point in time will be reached where the webserver will be waiting for the data that was sent over Path 2 to reach the laptop, so that it can send data again. This is because, until the data from Path 2 arrives on the laptop, it will not send an acknowledgement that allows the webserver to shift the window (mentioned in section 8.1.1) in which it can send new data.

This situation results in the MultiPath connection under-performing because it cannot keep sending new data over Path 1. Increasing the size of the receive window to the minimum requirement will allow the first path to be used again. If this was the case, the arrival of the data on Path 2 comes "early" enough so that the sender has not yet reached the limit of the window in which it can send data. Upon reception, the laptop sends an acknowledgement that results in the sender being able to shift the window in which it can send data by a big portion, resulting in the sender constantly being able to keep sending data. However, as said before certain devices are unable to allocate more resources to the connection, therefore providing a larger receive window is not an option. In this case a different solution must be used.

An example solution to this issue is also provided in [13]. Here, two mechanisms are introduced to compensate for delay differences between paths, and prevent the sender from waiting. The first mechanism, called Opportunistic Retransmission (OR), is applied when an available path (with low RTT) cannot send new data to the receiver due to receive window limitations. Here, the path is instead used to retransmit currently unacknowledged data to allow the receiver to acknowledge the data that was originally sent over the slower path in a quicker fashion. This allows for the sender to shift the window in which it can send data faster than when it had to wait for the data of the slower path to arrive at the receiver. The second mechanism, called Penalization (P), reduces the congestion window of the slower path to prevent this situation from happening often.

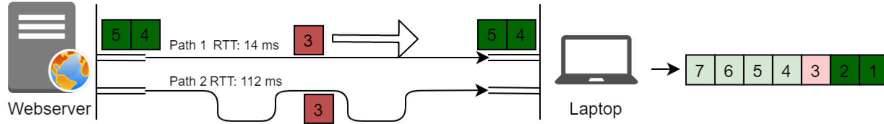


Figure 53: Retransmitting packet 3 over Path 1.

From the scenario's perspective: The same setup is used as in the previous scenario. A webserver is connected to a remote laptop via two paths, being Path 1 (with a low RTT) and Path 2 (with a higher RTT). The webserver wants to send data (a font-file) over a single stream in 7 sequential packets to the laptop via a Lowest-RTT-First scheduler that also applies ORP. The laptop has advertised a receive window that can receive three packets (for the sake of simplicity, a number of packets is used in all the windows instead of an actual number of bytes). The congestion window of Path 1 allows at most 2 packets to be sent, while the congestion window of Path 2 only allows a single packet to be sent. Thus, the webserver will prioritise Path 1 and send packets 1 and 2 over the path.

After this, the webserver cannot send packet 3 over Path 1 due to the congestion window being full, thus it sends the packet over Path 2. Upon reception of packet 1 and 2 the laptop processes them and sends an acknowledgement back to the webserver. The webserver is now able to shift the window in which it is allowed to send data past packet 2. Packets 4 and 5 can be sent over Path 1, but packets 6 and 7 cannot yet be sent by the webserver. This is because the laptop has not yet received packet 3, which is still being transmitted over Path 2. At this point, the webserver identifies that it is being throttled by the receive window of the laptop, and that packet 3 has not yet been received by the laptop. Therefore the webserver decides to apply the ORP mechanisms and retransmits packet 3 over Path 1, while the congestion window of Path 2 is halved. See Figure 53 for a visualisation.

The laptop then receives packet 3 over Path 1 and sends an acknowledgement to the webserver that allows the webserver to shift the window in which it can receive data past the fifth packet. At this point, the webserver is able to transmit packets 6 and 7 over Path 1. The laptop receives these two packets and sends an acknowledgement back to the webserver. After this, the laptop finally receives the third packet on Path 2. However, the laptop sees that the data of packet 3 was already received previously and discards the packet. If the webserver did not apply the ORP mechanisms in this situation, it had to wait for this packet to be acknowledged by the webserver before it could send packet 6 and 7 to the laptop.

In [40] multiple packet schedulers for MPTCP are compared and evaluated. In the paper it is concluded that the Lowest-RTT-First scheduler performs the best when ORP is enabled, compared to the other schedulers mentioned in the paper. An important note that is made in the paper is that a scheduler should aim for the data to be delivered over the multiple paths in an in-order fashion at the receiver. If a different (and worse) scheduling decision is made it can lead to situations where data is transmitted over high RTT paths, leading to the receiver having to wait for the data to arrive before it can be passed to the receiving application. Additionally, the server will then have to wait for the receiver to send an acknowledgement before it can send additional data.

8.2.3 MPQUIC packet scheduling

As was the case in MPTCP, there also have been multiple proposals for packet schedulers in MPQUIC. Initially, the same basic packet schedulers like Round-Robin could be used, but then again, the same problems could arise as in MPTCP. Thus, in QUIC it is also a better option to implement a packet scheduler that takes the characteristics of the paths into consideration. In comparison to MPTCP, there are some differences that can be identified, which simplify and/or slightly change how these packet schedulers work.

A first difference is the fact that QUIC makes use of frames that are independent of the packet (and the path) that carries them. In MPTCP, if a series of packets was accidentally sent over a path with a high RTT, and if the packets were dropped by a middlebox due to congestion, the sender was obligated to retransmit the exact same packets over the same path (even if the same data was retransmitted over a different path). This is to ensure that middleboxes on the path will not drop future packets that are transmitted due to their sequence numbers not matching the expected values. MPQUIC does not have to do this, since the data that is transported by the packets is independent of the packet, and the packets that are sent over a path will always have a monotonically increased packet number. This simplifies the retransmission mechanisms in the protocol.

A second difference is that, besides stream data, in MPQUIC additional control frames must also be scheduled over the available paths by a packet scheduler. While these control frames are also independent of the packet and path that carries them, it is an additional consideration that must be taken by a packet scheduler. For example, the MAX_STREAM_DATA-frame and the MAX_DATA-frame must be sent over all available paths to ensure that its delivery to the peer is as fast as possible. As said before, these frames are conceptually equal to the receive window of TCP, and therefore late delivery of these frames can lead to the same situation as in TCP, where the sender cannot transmit new data that surpasses these limits. Other control frames, such as the NEW_CONNECTION_ID-frame, can just be scheduled over an individual path.

In general, the functionality of a packet scheduler in MP QUIC remains the same as in MPTCP. Frames are placed in packets and the packets are then scheduled over the multiple available paths. A study [41] evaluated the performance of both MPTCP and MPQUIC against each other. In the paper both implementations used the same congestion controller, being OLIA, and the same packet scheduler, being Lowest-RTT-First. It must be noted that the MPQUIC implementation always provided increased stream limits and connection limits early enough to not encounter "receive window" limitations.

The paper concludes that the MPQUIC implementation performs better than the MPTCP implementation. The Lowest-RTT-First packet scheduler in MPQUIC was also expanded upon to support use cases where additional available paths currently have an unknown RTT estimate. In this case, data that is being transported on the other paths is duplicated and sent over this "unknown" path to allow the characteristics of that path to be formed. This initially introduces some overhead, but also prevents the packet scheduler from making uninformed decisions.

There is one more change that also must be considered, and that is the fact that QUIC supports the use of multiple (bidirectional) streams, compared to the support of a single stream in TCP. Previously, on a MultiPath connection a packet scheduler on a sending end-point would collect the data from a single input stream and decide how this data should be sent over the available paths. If this idea is expanded upon, and multiple streams are available, an initial idea could be to introduce a stream scheduler that is based on the Round-Robin mechanism. Here, data from one specific stream is taken to be placed inside a packet, and then that packet is scheduled on the best path according via the path scheduler. Every time a packet is sent, data from a different stream (in a looping manner) would be carried by that packet.

Figure 54: Round-Robin stream scheduling.

From the scenario's perspective: The webserver is connected to the laptop via two paths, being Path 1 and Path 2. Both paths have a similar RTT. the webserver has three available streams for which it must transmit data to the laptop. From each stream two packets have already been sent to the laptop. Assuming that the congestion controllers for each path allow for sending two additional packets, the webserver decides to send the next batch of packets to the laptop. Here, it first takes packet 3 from stream 1 and sends it over Path 1, since it has the lowest RTT. After this, it takes packet 3 from stream 2 and also sends it over Path 1. Lastly, the webserver takes packet 3 from stream 3 and sends if over Path 2, since the congestion window of Path 1 does not allow for another packet to be sent. See Figure 54 for a visualisation.

However, this method of scheduling stream data over the paths can potentially decrease the overall performance of the receiving application. For example, the laptop could be requesting the contents of a webpage, where stream 1 carries the data of a HTML-page, stream 2 the data of a CSS-file, and stream 3 the data of a JSON-file that is not as important as the first two files. The laptop needs the HTML-file and the CSS-file to generate the webpage in a browser application, but does not really need the JSON-file to do so. Therefore, stream 1 and 2 are more important and should be sent first by the webserver over the available paths. The current Round-Robin based stream scheduler does not take into consideration the characteristics of the given streams. Therefore, a different form of stream scheduling should be used that does take into consideration the stream characteristics. Looking at regular QUIC, the protocol supports multiple streams that a sending application can use to transmit data. In the protocol's design, the data from these streams are multiplexed via a prioritization scheme to improve the performance of the connection. Thus, if the three streams carried the same data as mentioned above, the data from stream 1 and 2 would have a higher priority and therefore is sent first, before the data from stream 3 is sent, which was given a lower priority. To allow the protocol to prioritize certain streams over others, it already provides the sending application with a way to indicate the priority that each stream has during the connection. Therefore, each stream is given a priority value between 0 and 255. The higher the priority value, the higher the importance of that stream is.

This mechanism, that gives each stream a certain priority value can also be used when a MPQUIC connection is established. Via stream prioritization, a stream scheduler could take data from streams that have a higher priority value to place them in a packet and allow the path scheduler to schedule the data on the best path. If there are multiple streams that have the same priority value, the stream scheduler will take turns in a Round-Robin manner. If a high priority stream is blocked by its flow-control limits, a stream with lower priority will be selected. With this mechanism, important data such as HTML-files and CSS-files will be prioritised and sent first.

Figure 55: Priority-based stream scheduling.

From the scenario's perspective: the same setup is used as in the previous scenario. However, a stream scheduler is used, and the three streams that are used by the connection each have their own priority value allocated to them. Streams 1 and 2 have a priority value of 255, indicating that these streams have the highest priority in the connection, whereas stream 3 has a priority value of 10, indicating that it has the lowest priority. Assuming (again) that the congestion controllers for each path allow for sending two additional packets, the webserver decides to send the next batch of packets to the laptop. Here, the stream scheduler sees that stream 1 and stream 2 both have the same, highest priority value, thus it generates a packet with data from each stream and gives the packets to the path scheduler. The path scheduler then sends the packets over Path 1, since it has the lowest RTT. After this, the stream scheduler again sees that both stream 1 and 2 have the highest priority value, and that neither stream is blocked by its flow control. Therefore, another packet is generated for each stream and given to the path scheduler. This time, the path scheduler sends the packets over Path 2, because the congestion window of Path 1 does not allow for another packet to be sent. See Figure 55 for a visualisation. The only way for data from stream 3 to be selected is when streams 1 and 2 both are blocked by their flow-control limits, or if all the data from those streams has been sent successfully.

A paper [42] has proposed a stream-aware scheduler that is based on this idea of prioritising streams that have a higher priority value. In the paper this scheduler is compared to other proposed schedulers for MPQUIC, and it concludes that the stream-aware scheduler provides better performance and reduced web-latency when using heterogeneous paths. It must be noted that the paper combines stream prioritization with an Earliest Completion First packet scheduler, which is different from the Lowest-RTT-first packet scheduler. Explained broadly, the Earliest Completion First packet scheduler also prioritizes paths that have the lowest RTT until it is blocked by its congestion window. At this point, the scheduler deviates from Lowest-RTT-First by not just selecting the next best path to send data on. Instead, the scheduler evaluates whether sending data over the "slower" path is beneficial. This evaluation is based on the path's RTT, along with how much data ls left to send and the throughput of the path. This could result in the scheduler deciding not to send data over the slower path and just wait for the faster path (and its congestion window) to recover.

By using a combination of stream schedulers and packet schedulers a MPQUIC connection can be further optimized to provide better (perceivable) application performance. However, further research is required in the scheduling of, and prioritizing of multiple streams in a MultiPath connection. For example, a different paper [43] gives the argument that the stream scheduler from the previous paper allows the prioritized streams to compete for the resources of the best path, causing burst transmissions that can degrade the throughput of that path. In response, the paper also proposes its own stream scheduler that includes more stream characteristics into consideration, such as the size of the data.

For the MPAIOQUIC implementation, only a dummy packet scheduler is implemented. A discussion about implementing a real MultiPath packet scheduler in MPAIOQUIC can be found in section 8.4.1.

8.2.4 Acknowledgement strategies

While it is not truly a part of the concept of packet scheduling, the way in which acknowledgements are sent by the receiver can also have an impact on the effectiveness of the sender's packet scheduler. In MPTCP, each subflow has its own acknowledgement mechanism that confirms the reception of data on the specific subflow. There is nothing that can be changed here as it is inherently a part of standard TCP. The DATA_ACK however, that acknowledges the reception of data on a connection level, can be freely transmitted by the receiver over the available subflows. Moving over to QUIC, the frames that are sent via packets over the available paths are independent of the packets and paths that carry them. Based on this, the (MP_)ACK-frames are also independent, and therefore the frames can also be sent over any of the available paths in the connection. It is perfectly possible that an endpoint has received an (MP_)ACK-frame on a given receiving uniflow (for example, receiving uniflow 1), where the frame acknowledges packets that were sent by the endpoint on a different sending uniflow (for example, sending uniflow 2).

However, this does not mean that acknowledgements should be transmitted randomly over the different available uniflows in the connection. This is because the endpoints measure and estimate the RTT values of each sending uniflow, and based on these measurements/estimates the congestion controllers and packet schedulers make decisions on how new data should be transmitted to the peer. If acknowledgements are sent randomly over the uniflows to the peer, the measured/estimated RTT values could differ greatly from reality, and as a result wrong decisions could be made by an endpoint that could lead to degraded throughput. Thus, a consistent strategy should be applied by the endpoints to ensure that RTT measurements/estimates are more accurate and closer to the real values. One such strategy is to send acknowledgements over a globally chosen sending uniflow for the duration of a connection, this chosen sending uniflow could be the initial sending uniflow. Another strategy could be to always sent (MP_)ACK-frames that belong to a certain receiving uniflow over the same chosen sending uniflow for the remainder of the connection, like mapping data from A via acknowledgements from B (These examples were mentioned in an earlier draft version of Q. De Coninck). Unlike the MAX_DATA-frames and MAX_STREAM_DATA-frames, the (MP_)ACK-frames should not be duplicated and sent over all the available sending uniflows, because these frames might introduce large network overhead.

8.3 Usefulness of MPQUIC

After the publication of the QUIC core protocol specifications, a virtual interim meeting was scheduled between various groups that were working with the QUIC protocol [24]. In this meeting, the goal was to discuss and understand the use cases that the groups envisioned when the concept of MultiPath is introduced to QUIC. From this interim, and from the discussion in the mailing list [44] that followed afterwards, multiple different use cases (and perspectives) could be identified on how MultiPath should be implemented in QUIC. For example, some use cases were limited to the use of just two (disjoint) paths in an end-to-end context, while other use cases expected some sort of (explicit) intermediary, such as a transport proxy or tunnel endpoint. The draft of Q. De Coninck used in this thesis focuses on the end-to-end context, where multiple paths can be identified on an endpoint via available IP-address combinations. As previously discussed, this draft is not the only proposal that was made for the implementation of MultiPath in QUIC. Additionally, and more in general, the ideas of how the concept should be implemented is even more spread out.

From the perspective of just sending packets, there are multiple classes of packet sending that could be used when multiple paths are available to the connection. A first class comes in the form of traffic steering, where only one path is chosen to send the packets on. A second class is given as traffic switching, where at a certain point during the connection, a different path could be chosen to send the packets on, while the use of the previous path is halted. A third class of packet sending called traffic switching, allows for the connection to send the packets over the multiple available paths simultaneously. Based on these three classes, a first discussion point can be identified. Some would argue that the current specification of QUIC already supports the concept of MultiPath, because the connection migration feature allows for traffic steering and traffic switching. Thus, the current mechanisms of QUIC could be enough. Even here, another discussion takes place, as some would argue that the path migration feature could be used to switch to a backup path when the main path becomes unavailable, while others argue that the feature can be used to frequently respond to changing path conditions. Coming back to the first discussion point, others would argue that the use of the path migration feature is only a lightweight form of MultiPath, and that further expansion should be implemented where multiple paths are used simultaneously for traffic switching.

Diving deeper into how traffic switching should be applied, another discussion point can be identified. During the interim meeting, various sending strategies were presented that could be used in a MultiPath QUIC connection, some of them already being implemented for MPTCP. As said before in section 8.2.1, applications might have different goals that need to be adhered to while sending data. Strategies that were presented to satisfy these goals are, but not limited to, strategies such as Active-Standby (where additional paths are used as backup when the main path becomes unavailable), Latency-Bandwidth tradeoff (where low latency paths of high bandwidth paths are preferred according to the application goals), and RTT-thresholds (such as the Lowest-RTT-First scheduler explained in section 8.2.3). Each of these strategies can be used for certain application goals, but might not be suited for others. Combinations of these strategies can also be provided to better suit the needs of applications. From the discussion, the argument is given that it would be better to start with basic scheduling schemes that would satisfy most application requirements.

Besides just scheduling packets over the available paths of a connection, the introduction of multiple streams in QUIC also adds another layer of complexity to the idea of traffic switching. Certain streams may carry data that is more important than the data of other streams, and prioritization should also be applied to scheduling stream data. Here, another discussion point can also be identified, because some argue that the scheduling of data from multiple streams should be done in QUIC, and thus, the transport layer. An example of this idea is already discussed in section 8.2.3, where as an example a priority-field could be added to each stream to indicate the importance of its data. Others argue that the scheduling of stream data could be given to the applications that use QUIC to transport the data (or even move the concept of MultiPath towards the application layer instead of the transport layer). Via an interface that allows QUIC to inform an application about the characteristics of a path, the application could make the scheduling decisions.

From one perspective, the choice could be made to provide scheduling in the transport layer, because in the past this was done to prevent having to change applications and requiring them to be aware of the multiple paths. From the other perspective, an application might be better off if it is indeed aware and could control the scheduling of data by itself. For example, as said in the interim meeting, if an endpoint provided video streaming, using the bandwidth from multiple paths could be useful. However, if rebuffering would be an issue, and if the endpoint wants to make sure that the video does not stutter, it would need to keep the risks of using the wrong paths in mind. If the endpoint cannot adjust or see the MultiPath operations that are happening underneath, it might not be able to ensure that it does the behaviour that it needs.

While all these discussion points occur on different levels, the argument that is given in general is that further experimentation on if and how people would use MultiPath must be done. Currently, it can not yet be said which approach to MultiPath is the best. Thus, it also can not yet be said that the draft used in this thesis is the most optimal way of implementing MP in QUIC. Having more implementations that support the same proposal or a different one allows for more and better comparisons to be made.

8.4 Future work

The implementation of the MultiPath concept in AIOQUIC provided us with the ability to perform tests and interops with the MP implementation in PQUIC. From these tests a series of problems have been identified and fixed (on both implementations), as discussed in section 7. However, due to the focus primarily being placed on path establishment and performing interops with PQUIC, other parts of the MP concept were left out in MPAIOQUIC. For the MultiPath congestion control and packet scheduling mechanisms only a dummy implementation was given that provided minimal functionality for the interop to work. In future versions of the MPAIOQUIC implementation, these dummy implementations are to be replaced with actual mechanisms that allow for the effectiveness of the design of Q. De Coninck to be evaluated in a broader spectrum. A discussion about the implementation of these mechanisms is given in section 8.4.1. Additionally, while implementing the concepts that were proposed in the draft, we noticed that the mechanism that uses UNIFLOWS-frames to signal IP-address usage could be simplified. A discussion about this is given in section 8.4.2.

8.4.1 Congestion control and packet scheduling in MPAIOQUIC

In the current implementation of MPAIOQUIC only a dummy congestion controller is implemented. Due to the focus of this thesis being the establishment of a successful interop with the PQUIC-implementation, less time could be spent on implementing a real MultiPath congestion controller. The provided dummy implementation is a placeholder that does not yet increase the congestion windows of a MultiPath QUIC connection, therefore limiting the sending uniflows to a fixed transmission rate supported by the initial value of the congestion window. In a future version of MPAIOQUIC, a real MP congestion controller should be implemented to allow better validation of the proposed MP mechanisms.

Since the regular AIOQUIC implementation already provides a New Reno-based congestion controller, a good first candidate for a MP congestion controller in MPAIOQUIC is OLIA (also based on New-Reno), which is already discussed in section 8.1. The dummy congestion controller mechanism already provides each sending uniflow with its own congestion controller and congestion window, along with the congestion windows of all the other sending uniflows in the connection. Besides this, the regular New Reno-based implementation of a congestion controller in AIOQUIC also provides functionality to allow the manipulation of the congestion window upon experiencing events such as receiving acknowledgements, triggering of losses and sending of data. Therefore, the implementation of OLIA only requires the writing of mechanisms to manipulate the congestion window via its additive increase/multiplicative decrease-scheme.

Besides the dummy congestion controller, a dummy packet scheduler was also provided in MPAIOQUIC. This packet scheduler is a Round-Robin based scheduler that takes the available paths of a connection and loops over them for each frame that needs to be sent to the peer. Additionally, AIOQUIC also only provides a Round-Robin based stream scheduler that loops over the available streams to generate STREAM-frames. In a future version of MPAIOQUIC, a real MP packet scheduler and stream scheduler should be implemented to allow better validation of the proposed MP mechanisms. Since the regular AIOQUIC implementation does not yet support stream prioritization, an additional priority-field must be introduced in the STREAM-class. Additionally, a good first candidate for a stream scheduler in MPAIOQUIC is a scheduler that prioritizes streams based on their priority-value. An example has already been discussed in section 8.2.3. Besides stream prioritization, a good first candidate for a packet scheduler in MPAIOQUIC is the Lowest-RTT-First scheduler, which is also already discussed in section 8.2.3. The dummy packet scheduler already has access to the congestion controllers of each available path, via the QuicSendingUniflow-object. Besides this, the scheduler also provides functionality to start and end the scheduler, to pass a selected sending uniflow to write frames, and to finish selected uniflows to prepare the packets. Thus, the implementation of the Lowest-RTT-First only requires the writing of a mechanism to manipulate the selection of a sending uniflow to pass and write frames to.

If both the OLIA congestion controller and the Lowest-RTT-First packet scheduler are implemented, an MPAIOQUIC setup can be provided that is similar to the one that Q. De Coninck et al. [41] used to evaluate their MPQUIC design. This setup can then be used to perform our own evaluations in different networks and allows us to compare our findings.

Such network setups can exist out of different configurations between MPAIOQUIC and PQUIC, where the paths and endpoints can have different characteristics. For example, endpoints could be a server or a client, running either the MPAIOQUIC or the PQUIC implementation, possibly enable MultiPath, have a certain number of IP-addresses available and prefer a certain maximum number of paths to use. Besides this, a certain number of bytes (small or large files) could be requested by the client. For the network itself a combination of links can be used where each link, and the connection between each link, could have different characteristics, such as RTT-time, available bandwidth, average packet loss and processing delays. Based on this, simulations can be run in a network emulator, such as NS-3. Here combinations of regular QUIC connections and MultiPath connections can be made, along with the characteristics of the endpoints, links and connections that can fluctuate over different values. If comparable values for these characteristics were to be used as in [41], we would expect similar results for the MPAIOQUIC implementation. Afterwards, steps could be taken where additional congestion controllers, packet schedulers and stream schedulers are implemented in MPAIOQUIC. From here, further comparisons between the results of different combinations can be made, and experiences could be shared with the QUIC Working Group.

8.4.2 Removing the receiving uniflows info section

During the implementation of the proposed mechanisms from the draft of Q. De Coninck et al., we noticed that the design and communication of IP-addresses is similar to that in MPTCP. However, due to the design for QUIC being different in general, an additional frame was introduced to allow endpoints to communicate to their peer which IP-addresses were being used by which sending and/or receiving uniflow. By using UNIFLOWS-frames that contained address IDs and uniflow IDs, both endpoints could notify their peer about changes that have occurred as a result of new IP-addresses being used, IP-addresses being lost, or uniflows migrating to different paths. Besides this, both endpoints could allow their peer to learn public IP-addresses for which initially a private IP-addresses could have been communicated via an ADD_ADDRESS-frame. The use of this UNIFLOWS-frame is discussed in section 5.6. However, upon further investigation of the mechanism, we discovered that the receiving uniflows info section from the UNIFLOWS-frame could potentially be unnecessary.

In the current design, an endpoint could learn a new IP-address from the peer in one of two methods. The first method is explicit, where an ADD_ADDRESS-frame is received, in which a (potentially private) IP-address and an address ID is communicated. The second method is implicit, where the peer sends a packet to the endpoint from an IP-address that the endpoint has not yet seen before. From the first method, the endpoint that received the frame will know the IP-address and the corresponding address ID, and the endpoint could use this IP-address to initiate an additional sending uniflow to send data to the peer. However, the issue with this method is that the communicated IP-address might be private.

From the second method, the endpoint will only know the IP-address from the peer, and not the corresponding address ID. This means that the endpoint is not allowed to use this IP-address to initiate an additional sending uniflow, since it does not yet know the address ID of that IP-address. To mitigate these issues, the UNIFLOWS-frame can be used. However, only the sending uniflows info section of that frame is used by the endpoint.

The receiving uniflow info section only contains information that the endpoint already knows, or can deduce from the information that is (implicitly) given by either the sending uniflows info section in the frame, the ADD_ADDRESS-frame and/or the REMOVE_ADDRESS-frame. If we define the possible scenario's where a peer could perform a specific action and send (some of) these three frames (ADD_ADDRESS, REMOVE_ADDRESS, UNIFLOWS), we get the following: (1) Using a brand-new sending uniflow. The peer decides to use a new sending uniflow. More specifically, one that was never used before, like sending uniflow 1 at the beginning of the connection after the handshake. (2) Migrating an active sending uniflow. The peer could migrate a sending uniflow to a different 4-tuple because the destination IP-address in the 4-tuple shows lower RTT-times across all 4-tuples. (3) Changing sending uniflow state to unused. The peer could decide to no longer use a given sending uniflow. (4) Reusing a sending uniflow. The peer could have lost an IP-address, and wants to reuse a sending uniflow that was bound to that IP-address. Reusing the uniflow results in it being bound to a new 4-tuple. (5) Losing active sending uniflows and receiving uniflows. The peer lost them due to loss of an IP-address.

Looking at the scenario's, in scenario 1 the receiving uniflows info section has no real use for the endpoint, since the receiving state of the peer does not change, only an additional active sending uniflows entry is added. The receiving uniflow info section also has no use in scenario 2, since no sending uniflows are removed from the active sending uniflows info section (and no changes to the state of the receiving uniflows were applied by the peer). Scenario 4 is the same as scenario 1, except for the reuse part, thus the receiving uniflow info section also has no use here. In scenario 5, the loss of a sending uniflow is implicitly communicated by the lack of that sending uniflow's entry in the sending uniflow info section. An endpoint could identify which IP-address that specific missing uniflow was using, and compare it to the IP-addresses that its own sending uniflows are using to send data to. If a match is found, that sending uniflow's state is then changed to unused.

The only case where the receiving uniflows info section might be useful is in scenario 3. This is because the peer decides to stop sending from a specific uniflow, and that could be understood by the endpoint as the peer losing its sending uniflow due to the loss of an IP-address (thus, scenario 5). In this case, the addition of the receiving uniflows info section could help the endpoint understand that the peer has not lost an IP-address, and instead only decided to stop sending from a specific sending uniflow. However, this scenario is not explained in the current version of the draft and should be handled differently. At the time of writing this thesis, the proposal to remove the receiving uniflow info section from the UNIFLOWS-frame is still being discussed [45]. As a result, the proposal is not yet part of any future versions of the draft.

9 Conclusion

Initially, a large part of connections between two endpoints were TCP connections that used a single, bidirectional path to exchange data. These connections were limited in the fact that, when one of the endpoints lost their connection to a local WiFi-network, the connection had to be shut down. Additionally, the throughput of a sender was dependent on the ever-changing transport capabilities of the intermediate links that had to propagate the packets. To address some of the issues that TCP had, a new extension in the form of MultiPath was introduced. The concept behind MultiPath was for a connection to utilize multiple available paths between the two endpoints to transport data. For MultiPath TCP, this concept was introduced in the form of using multiple TCP connections that were identifiable by the endpoint as being part of a MP connection. The extension was specifically designed to create a best-effort scenario where MP operations could be performed in the current state of the Internet. Since its introduction, a series of additional studies have been conducted to further improve the effectiveness of MultiPath connections in TCP. New forms of congestion controllers were designed and compared against other implementations, and the support for the MultiPath operations was also being implemented in multiple systems, including Linux and MacOS.

However, the default design of TCP also had a couple of issues that were not easily fixed. Such as the fact that HoL-blocking can occur because TCP guarantees that data is delivered in-order. Therefore, a new protocol with the name QUIC was designed to overcome these issues. During the design of the protocol, the focus was placed primarily on improving single path design via, but not limited to, the implementation of security and encryption for header values, providing the ability for the connection to migrate to different 4-tuples, the reduction of connection setup time and the handling of the TCP HoL-blocking problem. MultiPath operations were also envisioned to be included in the design of QUIC. However, it was planned to leave the implementation of the concept for a later version of the protocol.

While the implementation of MultiPath had been postponed, a number of research teams and companies already started to implement the concept as an extension for the current version of QUIC. While they all took different approaches to the implementation to the extension, the concept of MultiPath was split into three parts, the first one being the path manager that provided mechanism to establish additional paths. The second part is the congestion controller that aims to achieve fairness and applies traffic redirection. The last part is the packet scheduler that makes the decisions on how data should be transmitted over the available paths. For each of the parts, new proposals have been made. In this thesis, we decided to implement the mechanisms that were proposed in the draft of Q. De Coninck et al. in the AIOQUIC implementation. After providing a set of mechanisms to allow for MultiPath operations to function, we performed multiple interop tests in two phases. The first phase aimed at performing interops between our MPAIOQUIC implementation and other public endpoints that ran different implementations. Here, the goal was to evaluate whether our implementation behaved correctly when connections were established with other endpoints that did not support MultiPath operations. In the second phase, our aim was to perform interops with the PQUIC implementation that supports the MP operations via a provided plugin. The goal was to evaluate and test the behaviour of the implementations when multiple paths between the two endpoints are available.

For the first phase, the results allowed us to conclude that our implementation behaved correctly when MultiPath operations were not allowed. More precisely, our implementation produced the same results as the regular AIOQUIC implementation. For the second phase, the situation was a bit different. Here, the decision was made to perform interops on a local laptop, where two endpoints run different implementations. A series of combinations were made, such as enabling/disabling MultiPath, running a specific implementation, running a client or server and requesting a large or small amount of data.

In an iterative manner, test connections were performed where bugs could be identified and communicated to the developers of PQUIC. Afterwards, a fix was pushed and the next iteration of tests could occur. Via this manner we were able to identify numerous problems that prevented the connection from operating normally. However, due to time constraints and the iterative process of identifying and fixing bugs being a very slow process, we currently cannot conclude that both implementations are able to interact with each other in the way that was envisioned when proposing the mechanisms in the draft text. Besides this, the MPAIOQUIC implementation primarily focused on the evaluation of the path manager mechanisms. The implementation currently only provides minimal dummy congestion controllers and stream/-path scheduler mechanisms.

Reflecting on how the testing and evaluation process has taken place, a different approach to the implementation of the MultiPath mechanisms could have been taken. We first implemented many of the mechanisms that were proposed in the draft text to ensure that MultiPath connections could be established, and that data could be sent over multiple paths simultaneously. After this, our focus shifted largely to performing the interops and tests with the PQUIC implementation. Because of this approach, less time could be spent effectively on establishing connections with a PQUIC-based endpoint.

Instead of focusing entirely on implementing the proposed mechanisms, we could have brought forward the start of the interops with PQUIC implementation. In this approach, new mechanisms could be implemented in the MPAIOQUIC implementation and immediately be tested in interops, allowing for a more organic growth of the MPAIOQUIC implementation. In the end, our approach resulted in the MPAIOQUIC implementation not being fully implemented and/or tested. For example, the migration feature that allows sending uniflows to migrate to a different 4-tuple has not been fully validated. Besides this, the operation of sending MP_ACK-frames also has not been fully validated. However, interops with other implementations, and interops with the PQUIC implementation give a partial validation to migration and the MP_ACK-frame operations respectively.

Looking at the testing and evaluation of the implementations from a different perspective, the approach to implementing the MultiPath mechanisms in the PQUIC implementation could also have been done differently. For example, in our initial tests, it was clear that the developers used plugin code that took certain actions for granted. For example, the PQUIC endpoint did not take into consideration the maximum number of sending uniflows that the MPAIOQUIC implementation wanted to use for the connection. Instead, the PQUIC implementation assumed that the MPAIOQUIC implementation always wanted to use two additional sending uniflows.

Additionally, the update process where the code is changed to match the updated mechanisms in a new version of the draft had some issues. For example, in version 3 of the draft, the retire_prior_to-field had been introduced to the MP_NEW_CONNECTION_ID-frame. This additional field was not present in the frames that were generated by the PQUIC endpoint. Lastly, the PQUIC implementation does not include all the proposed mechanisms from the draft text. For example, the nonce computation, the handling of REMOVE_ADDRESS-frames and the handling of MP_RETIRE_CONNECTION_ID-frames are currently not present. An approach where the code is developed to be a "close" match to the proposed mechanisms would have been better. However, before we implemented MultiPath in AIOQUIC, the PQUIC implementation was the only implementation that supported the operations that were proposed in the draft text, thus certain shortcuts could be taken to provide a working implementation. Nevertheless, by providing a second implementation that supports the mechanisms proposed in the draft text of Q. De Coninck et al., we have allowed future versions of both implementations to be a better match of the MultiPath operations envisioned by the draft text.

Finally, we have also discussed the fact that there are multiple proposals on how congestion control and/or stream/packet scheduling should be done in a MultiPath environment. While the mechanisms for regular QUIC connections are progressing towards maturity, more eyes are being directed on implementing the concept of MultiPath in QUIC. Some discussion points argue on whether the current design of QUIC might be enough, while other focus on whether the responsibility of packet/stream scheduling really should be given to the transport protocol. Due to our implementation not having a real MP-capable congestion controller and stream/packet scheduler available, no further insights on these discussion points can be given. However, we agree on the fact that more testing and evaluations should be performed about if and how people want to use MultiPath in QUIC connections. Future versions of the MPAIOQUIC implementation should include real MP-capable congestion controllers and stream/packet schedulers that would allow us to collect experiences and assist with this vision.

List of Figures

1 Potential paths for MultiPath. . . . 4
2 An example TCP connection. . . . 6
3 An example MPTCP connection. . . . 7
4 Creating a MPTCP connection. . . . 9
5 An additional subflow handshake. . . . 11
6 Sending an ADD_ADDRESS, resulting in a new subflow. . . . 12
7 An example situation where two devices have the same address but are on different networks. . . . 13
8 An example subflow initiation where the phone connects to the wrong device. 14
9 Sending a REMOVE_ADDRESS, resulting in closing subflows 1 and 2. . . . 15
10 Reassembling packets from multiple subflows without data sequence number. 16
11 Transporting "Hello world" from the webserver to the laptop. . . . 18
12 An error occurs while transporting Hello world from the webserver to the laptop. 23
13 Closing a MPTCP connection. . . . 26
14 A simplified QUIC connection setup. . . . 30
15 A new connection ID being exchanged. . . . 31
16 A connection ID being retired. . . . 31
17 A QUIC connection migrating. . . . 34
18 An example MPQUIC connection. . . . 36
19 Creating a MPQUIC connection. . . . 38
20 An initial MPQUIC connection with two uniflows. . . . 39
21 An abstract overview of the possible uniflows. . . . 41
22 Representation of a MPQUIC connection via design 2. . . . 42
23 Representation of a MPQUIC connection via design 3. . . . 43
24 An additional uniflow establishment. . . . 46
25 An additional sub-connection establishment. . . . 48
26 An additional path establishment. . . . 49
27 Sending a ADD_ADDRESS frame. . . . 51
28 Sending a UNIFLOWS frame. . . . 52
29 An overview of all sending (and receiving) uniflows. . . . 55
30 Removing IP-address A2. . . . 57
31 An overview of all sending (and receiving) uniflows after losing IP-address A2. 58
32 The laptop performing a migration of sending uniflow 0. . . . 61
33 Transporting Hello world from the webserver to the laptop. . . . 65
34 Closing a MPQUIC connection. . . . 68
35 An AIOQUIC client. . . . 72
36 An AIOQUIC server. . . . 73
37 Interop results. . . . 82
38 An example interop setup. . . . 83
39 A frame parse error leading to connection termination. . . . 84
40 Decryption errors occurring. . . . 85
41 PING-ACK-loop occurring after packet loss. . . . 85
42 PQUIC server stops sending. . . . 86
43 PQUIC client crashing when generating MP_ACK-frame. . . . 86
44 PQUIC client incorrectly trying to validate a path. . . . 87

45 The window in which the sender can transmit data. 91
46 A packet's RTT of 74 ms. 93
47 A duplicate ACK occurring. 93
48 An ACK gap occurring. 94
49 Multiple connections sharing the resources of a bottleneck link. 96
50 Two connections sharing the resources of a bottleneck. 97
51 Two connections pooling the resources of a network. 98
52 Explicitly sending a packet over a path with high RTT. 104
53 Retransmitting packet 3 over Path 1. 106
54 Round-Robin stream scheduling. 108
55 Priority-based stream scheduling. 109

List of Tables

1 The webserver's perspective on remote addresses after ADD_ADDRESS. 51
2 The webserver's perspective on remote addresses after linking IP-address A2. 53
3 The laptop's perspective on remote addresses. 56
4 The webserver's perspective on remote addresses. 56
5 The laptop's perspective on remote addresses after losing IP-address A2. . . . 58
6 The webserver's perspective on remote addresses after losing IP-address A2. . 58

Listings

1 Creating multiple endpoints. 75
2 Passing a configuration to the QuicConnection-object. 76
3 Tracking the local and remote IP-addresses. 76
4 QuicConnection-class's frame handlers. 77
5 QuicConnection-class's sending and receiving uniflow classes. 77
6 Changing IP-address of the client in AIOQUIC. 82
7 Changing IP-address of the client in MPAIOQUIC. 82

References

[1] Larry Peterson and Bruce Davie. *Computer networks: a systems approach*. Morgan Kaufmann, sixth edition, 2020.

[2] Sally Floyd, Jamshid Mahdavi, Matt Mathis, and Dr. Allyn Romanow. TCP Selective Acknowledgment Options. RFC 2018, October 1996.

[3] Sally Floyd, Dr. K. K. Ramakrishnan, and David L. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, September 2001.

[4] David A. Borman, Robert T. Braden, and Van Jacobson. TCP Extensions for High Performance. RFC 1323, May 1992.

[5] Alan Ford, Costin Raiciu, Mark J. Handley, and Olivier Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824, January 2013.

[6] Olivier Bonaventure, Christoph Paasch, and Gregory Detal. Use Cases and Operational Experience with Multipath TCP. RFC 8041, January 2017.

[7] Costin Raiciu, Mark J. Handley, and Damon Wischik. Coupled Congestion Control for Multipath Transport Protocols. RFC 6356, October 2011.

[8] R. Khalili, N. Gast, M. Popovic, and J. Le Boudec. MPTCP Is Not Pareto-Optimal: Performance Issues and a Possible Solution. *IEEE/ACM Transactions on Networking*, 21(5):1651–1665, 2013.

[9] Q. Peng, A. Walid, J. Hwang, and S. H. Low. Multipath TCP: Analysis, Design, and Implementation. *IEEE/ACM Transactions on Networking*, 24(1):596–609, 2016.

[10] Yu Cao, Mingwei Xu, and Xiaoming Fu. Delay-based congestion control for multipath TCP. In *2012 20th IEEE International Conference on Network Protocols (ICNP)*, pages 1–10, 2012.

[11] Toshihiko Kato, Shiho Haruyama, Ryo Yamamoto, and Satoshi Ohzahata. *mpCUBIC: A CUBIC-like Congestion Control Algorithm for Multipath TCP*, pages 306–317. Springer, 06 2020.

[12] Ramin Khalili, Nicolas Gast, Miroslav Popovic, and Jean-Yves Le Boudec. Opportunistic Linked-Increases Congestion Control Algorithm for MPTCP. Internet-Draft draft-khalili-mptcp-congestion-control-05, Internet Engineering Task Force, July 2014. Work in Progress.

[13] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 399–412, San Jose, CA, April 2012. USENIX Association.

[14] Benjamin Hesmans, Hoang Tran-Viet, Ramin Sadre, and Olivier Bonaventure. A First Look at Real Multipath TCP Traffic. In *Traffic Monitoring and Analysis*, volume 9053, pages 233–246, 04 2015.

[15] Philip Eardley. Survey of MPTCP Implementations. Internet-Draft draft-eardley-mptcp-implementations-survey-02, Internet Engineering Task Force, July 2013. Work in Progress.

[16] Mat Martineau and Matthieu Baerts. Mutipath TCP Upstreaming, 2019.

[17] Viet Hoang Tran, Quentin Coninck, Benjamin Hesmans, Ramin Sadre, and Olivier Bonaventure. Observing real Multipath TCP traffic. *Computer Communications*, 94, 02 2016.

[18] Robin Marx. Head-of-Line Blocking in QUIC and HTTP/3: The Details. `https://calendar.perfplanet.com/2020/head-of-line-blocking-in-quic-and-http-3-the-details/#sec_http3`, December 2020.

[19] Daniel Stenberg. *HTTP/3 Explained*. gitbooks, 2018.

[20] Quentin De Coninck and Olivier Bonaventure. Multipath Extensions for QUIC (MP-QUIC). Internet-Draft draft-deconinck-quic-multipath-05, Internet Engineering Task Force, August 2020. Work in Progress.

[21] Qing An, Yanmei Liu, Yunfei Ma, and Zhenyu Li. Multipath Extension for QUIC. Internet-Draft draft-an-multipath-quic-00, Internet Engineering Task Force, October 2020. Work in Progress.

[22] Christian Huitema. QUIC Multipath Negotiation Option. Internet-Draft draft-huitema-quic-mpath-option-00, Internet Engineering Task Force, October 2020. Work in Progress.

[23] Christian Huitema. QUIC Multipath Requirements. Internet-Draft draft-huitema-quic-mpath-req-01, Internet Engineering Task Force, January 2018. Work in Progress.

[24] Marx and Kinnear. QUIC 2020-10-22 Interim Meeting Minutes, 2020.

[25] Jeremy Lainé. Aioquic. `https://github.com/aiortc/aioquic`, November 2020.

[26] Brett Cannon. Network protocols, sans I/O. `https://sans-io.readthedocs.io/`, 2016.

[27] Martin Thomson. QUIC Implementations. `https://github.com/quicwg/base-drafts/wiki/Implementations`, December 2020.

[28] Marten Seemann. QUIC Interop Runner. `https://interop.seemann.io/`, December 2020.

[29] Robin Marx, Maxime Piraux, Peter Quax, and Wim Lamotte. Debugging QUIC and HTTP/3 with Qlog and Qvis. In *Proceedings of the Applied Networking Research Workshop*, ANRW '20, page 5866, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Quentin De Coninck, François Michel, Maxime Piraux, Florentin Rochet, Thomas Given-Wilson, Axel Legay, Olivier Pereira, and Olivier Bonaventure. Pluginizing QUIC. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 5974, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Martin Duke. 20th Implementation Draft. https://github.com/quicwg/base-drafts/wiki/20th-Implementation-Draft, October 2020.

[32] Maxime Piraux and Tim Mesotten. Running pquic picoquicdemo with multipath plugin throws error. https://github.com/p-quic/pquic/issues/12, October 2020.

[33] Quentin De Coninck and Tim Mesotten. Fatal error when using -q <name>argument on picoquicdemo. https://github.com/p-quic/pquic/issues/14, October 2020.

[34] Quentin De Coninck, Maxime Piraux, and Tim Mesotten. Receiving mp_new_connection_id-frame for non-existing uniflow id. https://github.com/p-quic/pquic/issues/13, October 2020.

[35] Quentin De Coninck, Maxime Piraux, and Tim Mesotten. Incomplete frames being send during MultiPath-supported connection. https://github.com/p-quic/pquic/issues/16, October 2020.

[36] Maxime Piraux, Robin Marx, and Tim Mesotten. Interop with aioquic decryption issue. https://github.com/p-quic/pquic/issues/18, December 2020.

[37] Tim Mesotten. Inverse interop with aioquic packet reception issue. https://github.com/p-quic/pquic/issues/19, December 2020.

[38] Jana Iyengar and Ian Swett. QUIC Loss Detection and Congestion Control. Internet-Draft draft-ietf-quic-recovery-34, Internet Engineering Task Force, January 2021. Work in Progress.

[39] Bruno Yuji Lino Kimura and Antonio Alfredo Ferreira Loureiro. MPTCP Linux Kernel Congestion Controls. *CoRR*, abs/1812.03210, 2018.

[40] 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*, CSWS '14, page 2732, New York, NY, USA, 2014. Association for Computing Machinery.

[41] Quentin De Coninck and Olivier Bonaventure. Multipath QUIC: Design and Evaluation. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, page 160166, New York, NY, USA, 2017. Association for Computing Machinery.

[42] Alexander Rabitsch, Per Hurtig, and Anna Brunstrom. A Stream-Aware Multipath QUIC Scheduler for Heterogeneous Paths: Paper # XXX, XXX Pages. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, EPIQ'18, page 2935, New York, NY, USA, 2018. Association for Computing Machinery.

[43] X. Shi, L. Wang, F. Zhang, B. Zhou, and Z. Liu. PStream: Priority-Based Stream Scheduling for Heterogeneous Paths in Multipath-QUIC. In *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–8, Aug 2020.

[44] Spencer Dawkins. What to do about multipath in QUIC, November 2020. Message to the QUIC working group mailing list.

[45] Quentin De Coninck and Tim Mesotten. What is the use of the "Receiving Uniflow Info"-Section in the UNIFLOWS frame? https://github.com/qdeconinck/draft-deconinck-multipath-quic/issues/12, December 2020.