Applying FQ-CoDel For Packet Schedulers In Tunneled Transport Layer Access Bundling

Felix Andersson Johansson

Faculty of Health, Science and Technology

Subject: Computer Science

Points: 30 hp

Supervisor(s): Prof. Andreas Kassler and Prof. Anna Brunström

Examiner: Prof. Simone Fischer-Hübner

Date: 200615

Computer Science

Felix Andersson Johansson

Applying FQ-CoDel For Packet Schedulers In Tunneled Transport Layer Access Bundling

Master's Thesis

Abstract

The number of devices and internet traffic for applications connected to the internet increases continuously. Devices provide increasing support for multi-homing and can utilize different access networks for end-to-end communication. The simultaneous use of multiple access networks can increase end-to-end performance by aggregating capacities from multiple disjoint networks by exploiting multipath communication. However, at this current point in time, multipath compatible transport layer protocols or multipath support at lower layers of the network stack have not seen widespread adaptation. Tunneled transport layer access bundling is an approach that allows for all types of single-path resources to exploit multipath communication by tunneling data over a Virtual Private Network (VPN) with transparent entry points on the User Equipment (UE) and on the internet. Commonly, such adaptation utilizes a single queue to buffer incoming packets which pose problems with fair multiplexing between concurrent application flows while being susceptible to bufferbloat. We designed and implemented extensions to Pluganized QUIC (PQUIC) which enables Flow Queuing Controlled Delay (FQ-CoDel) as a queueing discipline in tunneled transport layer access bundling to investigate if it is possible to achieve fair multiplexing between application flows while mitigating bufferbloat at the transport layer. An evaluation in the network emulator, mininet, shows that FQ-CoDel can add mechanisms for an instant, constant, and fair access to the VPN while significantly lowering the end-to-end latency for tunneled application flows. Furthermore, the results indicate that packet schedulers, such as Lowest-RTT-First (LowRTT) that adapt to current network characteristics, upholds the performance over heterogeneous networks while keeping the benefits of FQ-CoDel.

Acknowledgements

I would like to offer my special thanks to my thesis supervisors Andreas Kassler and Anna Brunström for their invaluable support and guidance throughout this project. Thanks also to Jonas Karlsson for providing the required hardware and evaluation environment.

Contents

List of Figures

List of Tables

1 Introduction

Every year the number of devices and internet traffic for applications connected to the internet increases. Applications follow the trend of requiring more data communication and better reliability while maintaining low latency, e.g. real-time multimedia streaming. According to Sandvine a total of 58% of internet traffic is related to video streaming as of 2018 [1]. To cope with these demands, the underlying physical networks must continuously increase their capacity at a high rate which is not sustainable due to the high development rate and increasing requirement from applications.

Today it is commonplace that user devices such as smartphones support multi-homing, i.e. devices that can send and receive data from different access networks such as cellular networks or WLAN. Further extensions such as multipath communication enable a device to utilize both networks simultaneously thus achieving multi-connectivity. This allows for network capacity aggregation to increase the available bandwidth and improve reliability in case of network failure. However, multipath compatible transport layer protocols and lower layer solutions have not seen wide adaptation which poses problems for exploiting multipath communication. If end-nodes do not support the same multipath protocols multipath communication cannot be enforced.

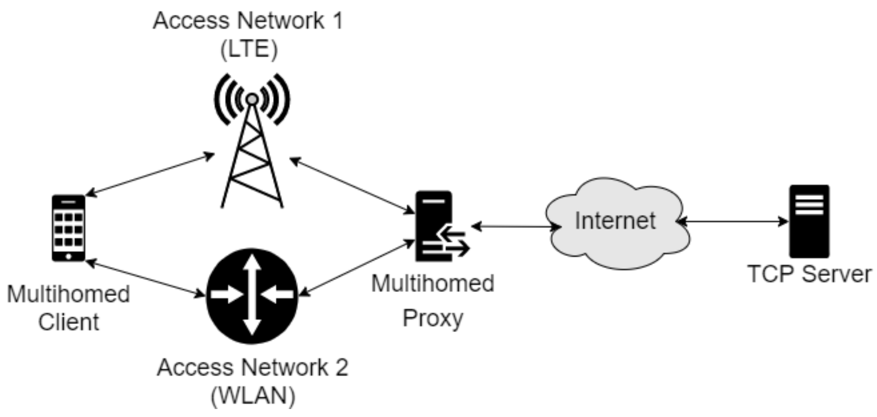


Figure 1.1: Client and Proxy with integrated access bundling communicating with a sending TCP server

Transport layer access bundling addresses the lack of multipath compatible resources by utilizing a transparent multi-homed proxy, an approach currently worked on by 3rd Generation Partnership Project (3GPP) for integration into 5G networks[2]. The principle is that a multi-homed client can establish a multipath connection to single path resources by sending data through a multi-homed proxy, see figure 1.1. The proxy incorporates functionality to map data from different paths into a single path connection and vice versa. The current approach by 3GPP uses Multipath TCP (MPTCP)[2] and suffers from a major limitation towards the increasing interest of real-time multimedia streaming[1]. The transport layer protocol enforces reliable end-to-end delivery of data which counteracts requirements towards timeliness. Thus, latency-sensitive applications cannot utilize multi-connectivity provided by MPTCP. Alternative transport layer bundling solutions such as MPTCP tunneling[3] attempts to address this by integrating a proxy in the client node, along with an external proxy. An MPTCP connection is established between the nodes to form a VPN through which any type of traffic can be tunneled. This allows for real-time traffic to exploit multi-connectivity at the cost of enforced reliable delivery of data inside the VPN. To remove the reliable delivery constrain alternative tunneling solutions exists that utilizes unreliable multipath transport layer protocols such as Multipath Datagram Congestion Control Protocol (MPDCCP)[4] and PQUIC[5]. PQUIC is an adaptation of the QUIC protocol[6] which allows for protocol extensions to be interchanged on a per-connection basis and supports multipath communication and unreliable delivery of data. PQUIC is also the framework used for evaluation in this thesis.

A common feature for tunneling solutions is to enable the VPN application to capture raw IP-packets from different application flows and buffer them in a single queue before they are encapsulated in the underlying multipath connection. This effectively multiplexes application flows at the transport layer and removes the ability to distinguish distinct flows at the lower layers of the network stack. Thus, performance gains from queueing disciplines at the link layer that supports fair multiplexing and requires a distinction between application flows, e.g. FQ-CoDel[7], are no longer working as intended. Furthermore, multiplexing different application flows into a single queue poses problems with fair-queue utilization between flows[7] and bufferbloat[8] if the queue is not managed properly.

This thesis explores if the end-to-end performance for application flows can be improved in tunneled transport layer access bundling by enforcing fair flow multiplexing and a low queuing delay in the VPN application. This includes a transport layer adaptation of FQ-CoDel[7] which maintains a separate queue for each application flow and per-queue Acitve Queue Management (AQM) to ensure a low queueing delay. Moreover, the thesis evaluates the interaction between the FQ-CoDel adaptation and different packet scheduling algorithms.

The contribution of this thesis are summarized in the list below:

- Open source implementation of an FQ-CoDel extension for PQUIC[9].
- An evaluation comparing the FQ-CoDel and FIFO queueing disciplines for tunneled transport layer access bundling.
- An evaluation of the interaction between FQ-CoDel and the packet schedulers Round-Robin (RR) and LowRTT

The thesis is structured as follows. Section 2 covers the necessary technical background, introduces transport layer access bundling solutions and problems with the existing approach. Section 3 introduces the framework, PQUIC, that is used to adapt FQ-CoDel and by allowing for advanced multiplexing to the transport layer. Furthermore, section 4 discusses the design and extends the motivation of why advanced multiplexing must occur at the transport layer. Following is a description of how FQ-CoDel is integrated into PQUIC and an explanation of the source code implementation. Section 5 describes how the adaptation of FQ-CoDel is evaluated and section 6 presents the results and evaluation. The thesis is finalized by a conclusion and proposed future work in section 7.

2 Background

To understand all the scope and goals of this thesis, it is important to cover the necessary technical background that it is based on. Knowledge about the fundamentals of computer networking is presumed and omitted. Instead, the section builds on understanding existing transport layer protocols solutions, their benefits, and drawbacks. This is extended by an explanation of how the knowledge is applied to improve network performance through exploiting Multipath (MP) communication. Following is an introduction to transport layer access bundling solutions and how they apply MP communication. The section is finalized with a discussion about queue management and how FQ-CoDel can address and improve the performance of the transport layer access bundling solution utilizing a single queue discipline.

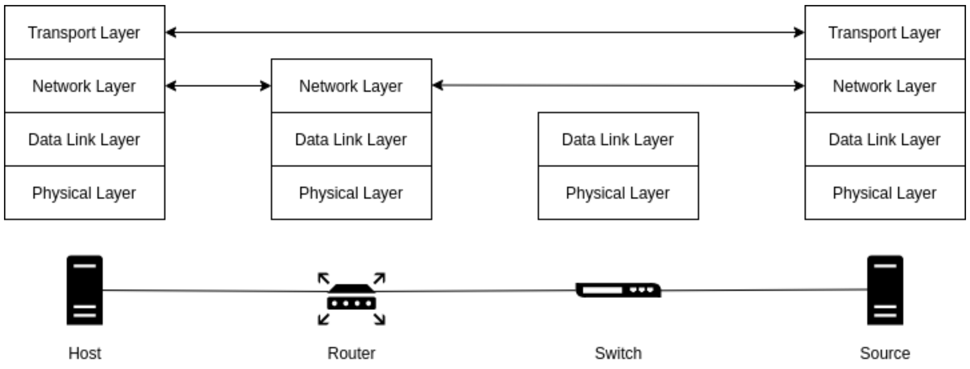


Figure 2.1: Logical connection between network layers and network nodes

2.1 Network Communication

Communication over the internet is based on the network protocol stack, or the OSI model[10], and the main design principle is enforcing separation of concerns through a layered architectural pattern. Each layer is contract-based with well-defined inputs and outputs to support interoperability between different systems and networks. As seen in figure 2.1, the different layers provide logical links between interconnected nodes and hosts.

Two of the most essential layers for end-to-end communication, and of most importance to this thesis, is the OSI model layer 3 and 4, see table 2.1.

OSI Layer	**Protocol**	**Mechanisms**
Layer 4/5	QUIC	- Connection-oriented - Reliable Delivery - Congestion-Controlled - Stream-oriented - Stream and Connection-level Flow Control - Stream Multiplexing - Connection Migration and resilience to to NAT Rebinding
Layer 4	TCP	- Connection-oriented - Reliable Delivery - Congestion-controlled - Stream-oriented - flow control
Layer 4	UDP	- Connectionless - Unreliable Delivery - Datagram Oriented
Layer 3	IP	- Stateless - Global Routing - Unreliable Delivery

Table 2.1: OSI Stack Layer 3-4

2.1.1 Internet Protocol Version 4

The main protocol of OSI layer 3 is the Internet Protocol version 4 (IPv4)[11]. The protocol supports logical interconnection between network nodes in packet-switched networks. The protocol provides mechanisms to enable unreliable end-to-end communication while remaining stateless. The Protocol Data Unit (PDU) are datagrams and provide functionality for fragmentation, reassembly, and addressing at a global scope to traverse interconnected heterogeneous networks. PDUs are globally routable from source to host through 4-byte IP addresses. More advanced mechanisms such as reliable delivery, sequencing, and congestion control must be defined by additional protocols in higher layers of the network stack.

2.1.2 User Datagram Protocol

User Datagram Protocol (UDP)[12] is designed to use IP as an underlying protocol and provides hosts with the ability to send datagrams with the same mechanisms as IP, see table 2.1. Therefore the protocol is connectionless and makes no guarantees for ordered or reliable delivery but operates with minimum overhead. The protocol is suitable for real-time applications that prioritize timely delivery. Due to UDP remaining connectionless it does not suffer from Network Adress Translation (NAT) rebindings as TCP does, see section 2.1.3.

2.1.3 Transmission Control Protocol

Due to the layered architecture of the network stack, it is possible to add protocols on top of lower layer implementations. The most commonly used layer 4 protocol is Transmission Control Protocol (TCP)[13] which provides mechanisms for reliable and ordered delivery of data, as a bytestream, on top of IP, see table 2.1. The protocol is suitable for file transfers due to the guaranteed in-order delivery.

An early design decision for TCP was to interlink TCP with the IP protocol, leading to a dependency between the transport and network layer. Datastreams and packets are distinguished by the common IP 5-tuple [1] which binds a connection to the IP addresses from the network layer. A change of IP addresses due to connection migration or NAT rebindings results in a connection teardown and reestablishment. Thus the continuously increasing number of mobile equipment, such as smart-phones and laptops, connected to an IP network presents a challenge for TCP.

2.1.4 QUIC

QUIC[6][14] is a connection-oriented, general-purpose transport layer protocol with similar mechanisms to TCP, see table 2.1. The intent of the protocol is addressing the shortcom-

[1] {source IP, destination IP, source port, destination port, and protocol}

ings of TCP by improving the quality of service. QUIC is designed to be implemented in user-space to ease distribution and adaptation compared to kernel implementations. Furthermore, QUIC uses UDP datagrams as a substrate for communication to avoid forcing changes in operating systems and to conform to the legacy functionality of hosts. UDP allows for traversal of middleboxes, e.g. NATs and firewalls, since the protocol has seen widespread adaptation, and QUIC is thus able to mitigate the network ossification problem [15].

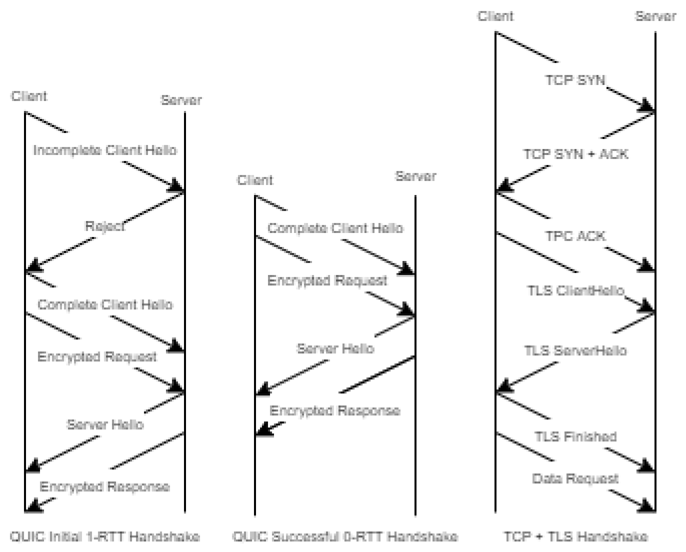


Figure 2.2: QUIC vs TCP Handshake

QUIC extends the available mechanisms available by UDP, see table 2.1, by encapsulating the additional data in datagram payloads. The encapsulated data and PDU of QUIC are packets and a packet consists of a sequence of one or more frames. The protocol achieves shorter connection establishment time compared to TCP by integrating TLS encryption into the handshake process making packets encrypted by default, see figure 2.2 for an equivalent comparison between QUIC and TCP + TLS. QUIC also provides a connection mechanism for resilience against connection migration and NAT rebindings. A connection is associated with a connection identifier instead of the IP/port 5-tuple used by

TCP. A connection can thus continue to associate and distinguish datastreams and packets when modifications to the IP 5-tuple occurs.

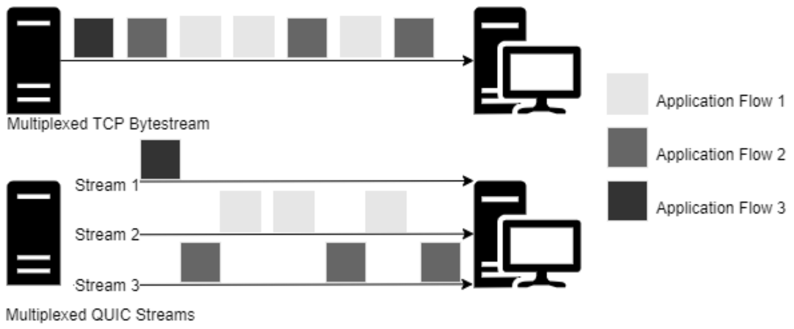


Figure 2.3: Application flow multiplexing differences between TCP and QUIC

A QUIC connection differs from TCP in the way multiple application flows can be multiplexed into and handled by a connection, see figure 2.3. In TCP, all application flows are multiplexed into a single bytestream where each arriving packet is sequenced and delivered in-order. A packet loss results in a time delay where all flows are temporarily blocked until the loss is resolved also called head of line blocking. QUIC solves this by introducing an abstraction layer to the multiplexing process where application flows are mapped into separate bytestreams within the connection. The separate bytestreams are referred to as streams and thus, a QUIC connection can consist of multiple concurrent streams. A packet can consist of QUIC frames from multiple different streams and each stream is independently marked with a byte offset to enforce in-order delivery. A packet loss will only cause a time delay and temporarily block to streams with frames in the packet that is lost.

Even though QUIC has addresses shortcomings of its predecessor, TCP, the protocol is not a good solution for all problems. As real-time multimedia streaming and other real-time applications are increasing in popularity the limitations of reliable and in-order delivery limited protocols become apparent. A protocol such as Datagram Congestion Control Protocol (DCCP) [16] attempts to address this through congestion-controlled unreliable delivery to ensure timeliness. However, this does not cover the use case where reliability is prioritized.

2.2 Multipath Communication

Above mentioned transport layer protocols are single network path solutions i.e. end-to-end communication is carried out from a single source IP address to a single host IP address. However, UE, such as smartphones, are commonly integrated with multihoming capabilities. These are devices with more than one unique IP address that can transmit and receive data on each of the addresses or otherwise referred to as network paths. E.g. a smartphone is commonly equipped with an IP address bound to a mobile network and an IP address bound to a WLAN network. The concept of exploiting MP communication stems from a single connection communicating over multiple network paths concurrently. This allows for aggregation of capacities of the different paths and presents them as a single resource, also called resource pooling[17]. Enforcing MP communication can increase throughput, improve resilience against network failure due to path failure, and increase reliability.

Exploiting concurrent MP communication introduces a set of new problems not applicable in a single path context. The connection must be able to detect and manage available network paths, distribute data over the paths and, remain fair to a single path connection. To solve these problems MP solutions must implement a collection of new modules and modify some old. The following modules and must be added or modified:

- Path manager
- Packet scheduler
- Congestion-controller

Upon connection establishment, the path manager is responsible for relaying information about peers' IP addresses and thus from which paths they are accessible. During a connection lifetime, the path manager can add or revoke paths available to the connection. The packet scheduler has access to the connections available paths and is responsible for distributing data over these paths. A packet scheduler should account for path asymmetry since different paths may experience different states of congestion, latency, and available bandwidth. Finally, the path scheduler and connection is rate limited by the congestion controller, similarly to a single path connection. Networks are shared mediums and all connections must ensure that the available capacity is fairly divided among all competing flows. An MP connection, using more than one concurrent path, may share a common bottleneck link with a single path connection resulting in unfair utilization. The MP congestion controller must be modified to ensure that the allowed transmission rate of all available paths is fair against single path connections.

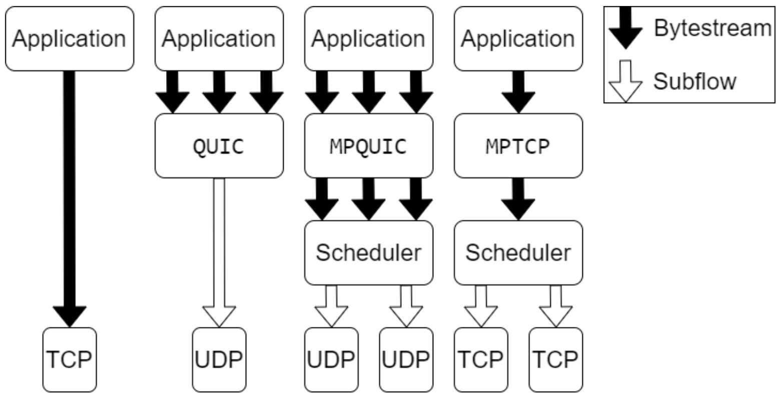


Figure 2.4: Architectural overview of differences between TCP, QUIC, MPTCP and, MPQUIC

2.2.1 Multipath Transmission Control Protocol

One of the first multipath protocols to emerge was MPTCP [18]. The protocol extends mechanisms of a TCP connection to send data over multiple network paths concurrently, achieving resource pooling. An MPTCP connection still consists of a single bytestream while maintaining a collection of one or more subflows (paths) on which data can be scheduled, see figure 2.4. A subflow is similar to a regular TCP connection, identified by the common IP 5-tuple, and requires a three-way handshake for connection establishment. The MPTCP connection is bound by a unique token, included in the handshake process, making it possible to distinguish subflows. This makes an MPTCP connection resilient to connection migration and NAT rebindings as one path may lose connection and be reestablished as data flows over different paths.

To ensure in-order reliable delivery to applications, MPTCP uses packet sequencing similar to TCP. But, to avoid problems with middleboxes, due to sequence gaps, each subflow must maintain a distinct data sequence that is mapped to the connection sequence. This design limits the connection flexibility since data sent over a specific path must be retransmitted and acknowledged over the same subflow. This effectively segregates subflows from each other and limits the packet scheduling possibilities.

Since MPTCP can utilize multiple concurrent paths the congestion controller must be coupled and encapsulate all subflows to maintain fairness against single path connections. MPTCP can apply many variations of congestion controllers such as Balanced Linked Adaptation Congestion Control Algorithm (BALIA)[19] or Weighted Vegas (wVegas)[20] to avoid congesting the network. An algorithm that ensures fair bottleneck utilization against a single path connection is Opportunistic Linked-Increases Algorithm (OLIA)[21] but, understanding the design of such algorithms is outside the scope of this thesis.

2.2.2 Multipath QUIC

The QUIC protocol is designed as a single path protocol with mechanisms to change the active path over a connection lifetime through connection migration. However, as mentioned before, this mechanism is designed as a failover measure to improve connection reliability rather than enforcing resource pooling and path aggregation.

De Conick et al[22] provides an extension, Multipath QUIC (MPQUIC), for QUIC that adds support for concurrent MP communication similar to MPTCP. Due to the extension being built on top of QUIC the extension can utilize and extend upon the base protocol features, see table 2.1. This enables the use of stream abstractions, stream multiplexing, 0-RTT connection establishment, etc. The implementation further extends upon that frames must be independent of the packet that carries them, by also enforcing that frames are independent of which path they are sent, see figure 2.4. Frames can thus be retransmitted and acknowledged on different paths.

MPQUIC is initiated as a regular QUIC connection, see figure 2.2, and during the negotiation process, the host and client announces and agrees on the number of paths that can be used during the connections lifetime. When the connection is established a new path can be added using a 0-RTT handshake and thus allowing data to be transmitted immediately. A path is defined by a common 5-tuple[2] and a unique identifier. Since a 5-tuple can be modified due to NAT rebindings or equivalent, a path must support reliable migration support to conform with QUICs connection migration mechanism. An MPQUIC connection thus supports path migration, i.e. migrating a path to another path instead of migrating the entire connection to a new path.

MPQUIC applies a modular congestion controller and must due to multi-connectivity maintain a congestion window per active path. Furthermore, to ensure fairness against a single path connection MPQUIC enforces a coupled congestion controller encapsulating the per-path congestion window. The current adaptation of MPQUIC supports OLIA[21] congestion control algorithm.

2.2.3 Scheduling Packets Over Available Network Paths

To achieve resource pooling in an MP context and maximize the gain for utilizing multiple concurrent paths it is important to consider how the packets are distributed over the available paths. Different paths may express different network characteristics, e.g. available bandwidth and experienced latency, and must be considered by the packet scheduler to

[2]{source IP, destination IP, source port, destination port, and protocol}

yield end-to-end performance gains. Packet schedulers are limited by a per-path congestion window and when all are filled the connection becomes ack-clocked. This behavior only allows the packet scheduler to schedule data on non-congested paths[23]. The section introduces different packet scheduling approaches that are applied in existing multipath compatible transport layer protocols such as MPTCP and MPQUIC.

Round-Robin scheduling is designed to achieve fair data distribution over a set of symmetric network paths. For each transmission opportunity, the packet scheduler cyclically alternates between the available paths, similar to any generic Round-Robin algorithm. The method of distribution makes it possible to utilize the aggregated available capacities of all paths and achieve resource pooling. To cope with asymmetric networks with different path capacities a weighted RR packet scheduler should be applied. RR or weighted RR packet scheduling is optimal for network paths with symmetrical latencies since it does not account for differentiating latencies when making scheduling decisions. Heterogeneous network paths will cause out-of-order delivery and introduce reordering delay if the transport protocol enforces reliable delivery or connection quality loss for protocols that support unreliable delivery.

Lowest-RTT-First [23] is designed to transmit data over the path with the smallest measured RTT. Once the prioritized path becomes congested, data is scheduled on the path with the next highest measured RTT. By prioritizing the path with the lowest RTT makes this approach more suitable for heterogeneous network paths. However, a long-living connection and bulk transmission will cause all paths to eventually become congested and cause the connection to become ack-clocked[23].

LowRTT is the default scheduler used in the MPTCP implementation in Linux and MPQUIC. However, the implementation differs due to the limitations of the protocols. Due to MPTCP subflows restrictions, data sent on a specific path must be acknowledged and retransmitted on the same path. Paash et al.[23] propose modifications to the LowRTT scheduler by introducing Retransmission and Penalization (RP), a method for MPTCP to minimize the limitations of its subflows. The scheduler can insert data that causes the head-of-line blocking into a different subflow. The MPQUIC adaptation of LowRTT allows for data to be acknowledged and retransmitted on any path regardless of the path initially used by default. Thus the optimization required in MPTCP already exists in MPQUIC. Common for both adaptations is that paths with high RTT are penalized by a reduction in the congestion window to reduce the impact of bufferbloat, see section 2.4.

There are many packet scheduling approaches, Hurtig et. al.[24] presents the behavior from low latency packet schedulers such as Delay-Aware Packet Scheduler (DAPS) and Out-of-Order Transmission for In-Order Arrival Scheduler (OTIAS), etc. A further approach such as Cheapest Pipe First (CPF) prioritizes the cheapest path depending on monetary cost of using a path, e.g. UEs prioritizing WLAN over mobile networks. The thesis focuses primarily on the evaluation of LowRTT and RR packet scheduler which is why they are covered in more detail.

2.3 Transport Layer Access Network Bundling

Transport layer access bundling solutions can be considered as a subset of multipath communication. An MP connection uses multiple paths between the sender and receiver and does not assume the underlying technology or type of access network used, e.g. LTE, Fiber, etc, except that they are IP based. This effectively bundles multiple heterogeneous access networks. However, far from all hosts has support for MP compatible transport layer protocols which prohibits MP connectivity from being a generic solution.

2.3.1 Transport Layer Access Bundling in 5G New Radio

Transport layer bundling is an ongoing research field for 5G multi-access worked on by 3GPP which enforces Access Traffic Steering, Switching & Splitting (ATSSS) to improve end user experience. ATSSS is defined as follows:

- **Steering**: Selecting a new access network a data flow should use to achieve load-balancing.
- **Switching**: Migrate an ongoing data flow to another access network as a failover measure in case of path quality degradation.
- **Splitting**: Split data flows over multiple access networks to achieve resource pooling.

ATSSS can be implemented as a transport layer solution in combination with ATSSS-lower layer. ATSSS-lower layer performs procedures for ATSSS over different 3GPP technologies, e.g. LTE and 5G, and is outside the scope of this thesis. The goal of the transport layer ATSSS is bundling 5G new radio together with non-3GPP based networks, e.g. WLAN, and making it an integral part of the 5G architecture[2]. The proposal utilizes MPTCP as a transport layer protocol and a transparent multi-homed proxy with an architectural design shown in figure 1.1. Both the UE and proxy must support MPTCP to enable multi-connectivity. The design is that a UE establishes an MPTCP connection to the proxy node over a 3GPP and non-3GPP network. The UE uses the MP connection to request data from a TCP compatible resource through the proxy node. The proxy incorporates functionality to map the incoming request from an MPTCP connection into a single path TCP connection which is forwarded to the destination. The reversed logic is applied when the proxy receives data from the resource. This allows for ATSSS functionality such as seamless handover between 3GPP and non-3GPP, load-balancing, and traffic steering with regards to the path with the lowest RTT[2]. The MPTCP protocol enforces reliable delivery of data and is limited to TCP compatible resources. This poses problems with increasing utilization of protocols such as QUIC and the increasing demand for real-time applications, e.g. real-time media streaming[1]. To allow for such protocols and applications to take advantage of transport layer access bundling, as proposed by 3GPP, additional protocols must be supported.

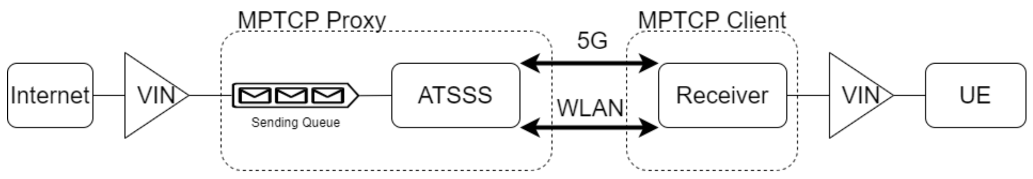


Figure 2.5: Architectural design of MPTCP tunneled transport layer access bundling

2.3.2 Tunneled Transport Layer Access Bundling

Liu et al.[3] propose using multipath compatible VPN solution utilizing MPTCP to counteract the limitations with 3GPP transport layer access bundling, see figure 2.5. The principle is to host a proxy entry point at a multi-homed UE and another located on the internet. A VPN tunnel is established between the entry points, over a 3GPP and non-3GPP network, using an MPTCP connection. Data is requested by the UE through a virtual interface and is captured at the IP level by the VPN application which allows for any IP based traffic to be tunneled. The VPN encapsulates the incoming data packet into the MPTCP connection by adding an MPTCP header to the captured data and appending it to a sending queue. Upon receiving data, MPTCP de-encapsulates the packet and forwards it to the egress virtual interface. Data that has left the VPN tunnel is routed to its destination according to the IP-header. A UE can thus request data from a single path resource through the VPN and take advantage of MP connectivity. The tunneled MPTCP adaptation allows for equivalent transport layer ATSSS functionality as described in section 2.3.1. It is important to reiterate that MPTCP is limited by in-order reliable delivery which may cause head-of-line blocking and is not suitable for real-time applications that do not require in-order, reliable delivery.

UEs utilizing a tunneled MPTCP access bundling solution allows for multiple concurrent data flows from different applications multiplexed into a single MPTCP bytestream, see figure 2.4. The flows are encapsulated and added to an unmanaged single First In First Out (FIFO) queue from which the packet scheduler operates. The scheduler is unable to distinguish between the different flows and makes scheduling decisions solely on path characteristics. The queue is a shared medium and it is important to ensure fair utilization and access between the flows. However, due to the wide variety of data flow (e.g. bursty, low rate, and long-lived), fair utilization is improbable without active management. The effect is high packet delay variation (jitter) due to varying queuing sojourn time and slow convergence to fair utilization further explained in section 2.4.

Amend et al. [4] proposes an alternative to the MPTCP access bundling based on a multipath extension to DCCP called MPDCCP. The prototype provides similar properties to MPTCP but provides support for unreliable data delivery over multiple paths. The approach is more suitable for low latency services and removes problems with head-of-line blocking and subsequent delays caused by it. The architectural design is similar to that of MPTCP access bundling, see figure 2.5, and data sent through the MPDCCP tunnel is passed through a virtual interface. The data is encapsulated and sequenced in order of arrival[4]. Because of this, there is no support for the packet scheduler to distinguish unique data flows, and performance will suffer from the same queuing problems mentioned above.

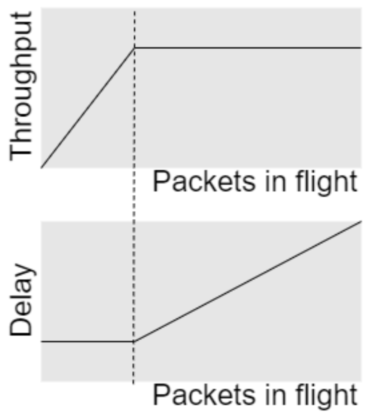


Figure 2.6: Data flow experiencing bufferbloat due to over dimensioned network buffers

2.4 Queue Management

The unmanaged single queuing approaches used by MPDCCP and MPTCP access bundling, mentioned in section 2.3, poses several problems that will result in degradation of performance. Tunneled transport layer access bundling is used to tunnel data from external applications and is considered a part of the underlying network from an application perspective. Queues and buffers are necessary to absorb temporary bursts of traffic but if implemented incorrectly can contribute to what is known as buffebloat[8]. Bufferbloat is caused by over-dimensioned buffers in the network that leads to excessive buffering and increased delay, see figure 2.6. The experienced throughput for a given flow, e.g. TCP, will increase until it matches the bottleneck link of the network and when this point is reached excessive packets will be buffered. If the buffers are appropriately dimensioned they absorb the surge in traffic and maintain a steady dequeuing pace thus forming a good queue. Large buffers can absorb large surges of traffic and introduce a buffering delay due to the rate limit of the bottleneck link. When a packet loss occurs, due to e.g. buffer overflow, its detection will be delayed by the buffering delay resulting in a less responsive connection. When a flow achieves a steady-state the buffer causing bufferbloat will maintain a standing queue due to the backpressure from the connection and the rate limit of the bottleneck thus forming a bad queue. The effect is a constant throughput and increased delay due to the queueing delay (queuebloat).

2.4.1 Active Queue Management

One attempt to solve bufferbloat is AQM[25] that attempts to address over-dimensioned buffers located at the bottleneck. Nichols et al.[26] proposes Controlled Delay (CoDel) an AQM algorithm that attempts to enforce a low queuing delay and allow for traffic surges, i.e. differentiate between good and bad queues and treat them differently. It operates from local metrics only and is insensitive to changing network conditions such as variable link rates and RTTs. A CoDel queue is FIFO ordered and every packet enqueued into CoDel is timestamped which allows the algorithm to keep track of packet sojourn time, i.e. the elapsed time between enqueuing and dequeuing. From CoDels' perspective, the queue is monitored using the sojourn times instead of using regular byte tracking or queue size. CoDel operates using two primary variables i.e. target time which is the targeted sojourn time for a given packet and interval time to track when packets can be dropped (head drop). Furthermore, CoDel considers a queue to be good if all packet sojourn times are below the target limit. When the targeted sojourn time is exceeded, e.g. due to dynamic links, for at least an interval the queue is considered bad and the algorithm drops a packet and enters the dropping state. Packet drops occur according to a control law by setting the next drop time to the"inverse proportion to the square root of the number of drops"[26] and continuous until the packet sojourn time is below the target. For a pseudo-code see [27].

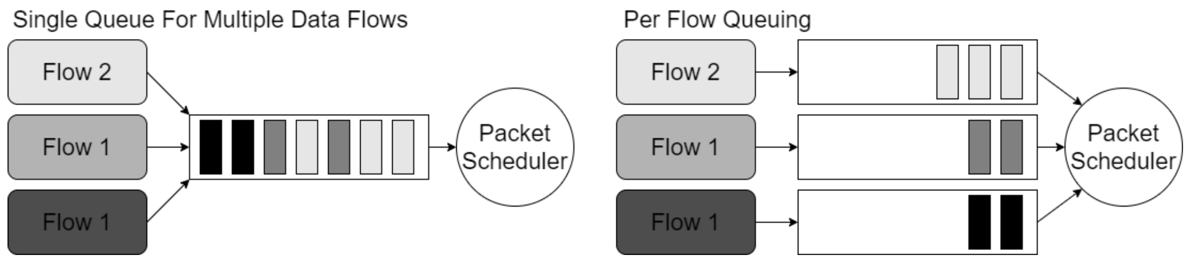


Figure 2.7: Depicting difference between multiple flows sharing a single queue vs a flow queuing approach

2.4.2 Flow Queuing

When multiple data flows share the same queue a problem arises with fair access, see figure 2.7, and all flows suffer due to the backpressure on the queue. Data from competing flows will be blocked by each other introducing a probabilistic queuing delay similar to head-of-line blocking. The solution, also shown in figure 2.7, introduces flow queuing. Each data flow is classified into separate queues providing isolation between them. A queue manager can thus ensure fair access to the network by implementing a RR algorithm that cyclically dequeues data from the different flows.

2.4.3 Flow Queue Controlled Delay

To address problems with both bufferbloat, see section 2.4.1 and single queue systems, see section 2.4.2, Hoeiland-Joergensen et al.[7] proposes FQ-CoDel. The approach is a multi-queue system to allow for flow queuing where each distinct queue is managed by CoDel. Furthermore, FQ-CoDel distinguishes between new flows (empty queues that just had data enqueued) and old flows (standing queues) by maintaining separate queues for the two. The different flows are defined as follows:

- **Flow**: FIFO ordered queue managed by CoDel, containing data from a specific data flow
- **New Flows**: FIFO ordered queue of queues containing data flows that cannot maintain a standing queue
- **Old Flows**: FIFO queue of queues containing data flows maintaining a standing queue

To accomplish flow queueing, data packets are captured at IP level to expose the IP header from which the common IP 5-tuple[3] is accessible. The 5-tuple is hashed to classify which queue the packet belongs to. A hash collision can occur which is manifested as two flows sharing a single CoDel queue.

Enqueueing The enqueueing procedure consists of classifying the incoming packet to a CoDel queue and timestamp it. A packet that is classified into an empty CoDel queue is considered as a new flow and the queue is added to *New flows*. If the packet is classified into a non-empty queue no action is taken by FQ-CoDel. Upon enqueueing the packet is handed over to CoDel, see section 2.4.1.

[3]{source IP, destination IP, source port, destination port, and protocol}

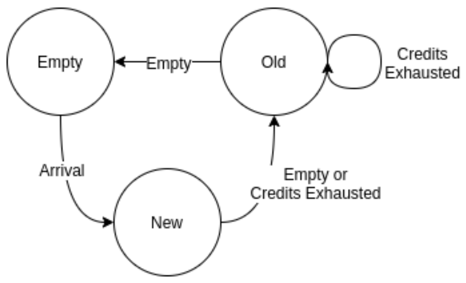


Figure 2.8: FQ-CoDel flow states

Flow Management A flow maintains a credits variable that corresponds to the allowed bytes to be dequeued per dequeuing opportunity. The value is given by the quantum variable equal to the Maximum Transmission Unit (MTU) of the link-layer interface. When a packet is dequeued from a flow its credits are reduced equal to the number of bytes of the dequeued packet. A queue can thus be considered for multiple dequeue iterations as long as it has credits. If a flow exhausts all credits it is moved to the back of *Old flows* and given a set of new credits equal to the quantum, see figure 2.8.

Dequeuing FQ-CoDels' dequeuing procedure is managed by a two-layer Deficit Round-Robin (DRR) algorithm to enforce flow queuing where *New flows* are considered first and *Old flows* after, see algorithm 3 in section 4.4 . The flow queuing approach does not enforce fair network access between competing flows due to the DRR prioritization and flow credits limit. Bursty traffic that is unable to maintain a standing queue will be considered as new flows and thus a higher priority in the DRR and cause starvation of long-lived old flows. To avoid starving old flows, a new flow that fails to dequeue a packet before it has exhausted all credits while there exist *Old flows* will be moved to the back of *Old flows* queue. A packet that is dequeued is managed by the CoDel dequeue algorithm.

Existing Implementation FQ-CoDel is currently implemented into the Linux kernel and is the default link-layer queueing discipline for several distributions[7]. The algorithm has proven to increase the performance in regards to fair share link utilization between flows and reduced end-to-end flow latency when benchmarked against the previous default single queue FIFO discipline. The Bufferbloat community provides statistics to support this claim through benchmarking the performance of FIFO compared to FQ-CoDel[28] where the tests consist of multiple concurrent flows sharing a bottleneck link. The performance increase strengthens the hypothesis that lifting FQ-CoDel functionality to tunneled transport layer bundling solutions can yield similar performance improvements.

3 Pluganizable QUIC

De Coninck et al.[5] presents the PQUIC framework that will be used for applying FQ-CoDel for tunneled transport layer access bundling. The framework is an implementation of the QUIC protocol that in its basic form conforms with the IETF QUIC specification and provides a multiplexed general-purpose transport protocol with guaranteed reliable and ordered delivery, see section 2.1.4. The version of PQUIC that is used in this thesis follows the draft-14 of QUIC. PQUIC extends the capabilities of a regular implementation by allowing multiple protocol extensions to be added, shared, or removed on a per-connection basis. This allows for a completely configurable transport layer protocol. PQUIC provides protocol extensions to support unreliable delivery of data, see section 3.4, and multipath communication, see section 3.5. Furthermore, the framework is shipped with a VPN application, see section 3.4, that when combined with the extension for unreliable delivery and multipath provides a transport layer bundling solution similar to MPDCCP, mentioned in section 2.3.1 and figure 2.5.

3.1 Framework Architecture

The PQUIC framework is based on an existing QUIC implementation, picoquic[29] that is written in C. The implementation is extended by allowing PQUIC to execute all protocol extensions inside of a virtual machine, the Pluglet Runtime Environment (PRE). The PRE provides memory separation between protocol extensions and different picoquic connections. This enables PQUIC to support multiple concurrent plugins per connection and different protocol extensions for each active connection. Protocol extensions are called *Plugins* and are named according to their capability of adding or removing a certain protocol feature. E.g the plugin named *disable_congestion_control* will disable a connections congestion control functionality and tracking for all communication on all network paths. A plugin is constructed using a set of one or more building blocks, referred to as *Pluglets* and a plugin *Manifest*. The manifest is what globally defines a plugin and how its pluglets are to be fitted into the execution workflow of PQUIC. Pluglets are platform-independent bytecode that is executed inside the PRE and whose functionality must conform with an existing protocol operation, see section 3.2.

3.2 Protocol Operations and Pluglets

Protocol operations are anchor-points for interchangeable sub-routines which are integrated into the frameworks execution workflow. Every protocol operation is designed for a specific purpose and PQUIC supports generic sub-routines for each protocol operation to ensure that a connection can be established and work as intended. An example of a protocol operations is *write_frame*, i.e. constructing a picoquic frame. A plugin can change the behavior of specific protocol operation by implementing a pluglet and use the plugin manifests to indicate which protocol operation that is affected, see algorithm 1.

Algorithm 1 Plugin Manifest

```
1: be.creator.pluginX                                        ▷ Name of plugin
2: protocol_operation_A replace pluglet_new_operation_A
3: protocol_operation_B param 0x01 replace pluglet_new_operation_B
```

When the plugin is inserted into a picoquic connection, PQUIC will read the plugin manifest and locate the *replace* keyword. This replaces the generic sub-routine in PQUIC with the new pluglet sub-routine. When the anchor-point is reached during execution the PRE will be invoked and the new sub-routine will execute. A protocol operation can also be parameterized which allows for coexistence between a generic sub-routine and a pluglet. A replacement action with a given parameter will only replace the sub-routine given that specific parameter. The parameter is the identifier for a picoquic frame which allows for the introduction of new frame types and different processing procedures of these frames. E.g the datagram plugin exploits this by introducing new frames that are processed to be non-retransmittable thus enforcing unreliable delivery of only these frames. There are a total of four different hooks that a pluglet can use to modify the behavior of a protocol operation where all can be parametrized to only affect specific frames:

- **Replace**: Replaces the default sub-routine
- **Pre**: A pluglet defined sub-routine executing before the protocol operation is invoked
- **Post**: A pluglet defined sub-routine executing after the protocol operation is done
- **External**: Plugin specific sub-routines that are callable from external applications and are accessible when the plugin is injected, see section 3.4 how this is used.

At the point of writing the thesis, there are a total of 72 different protocol operations available in the framework[5].

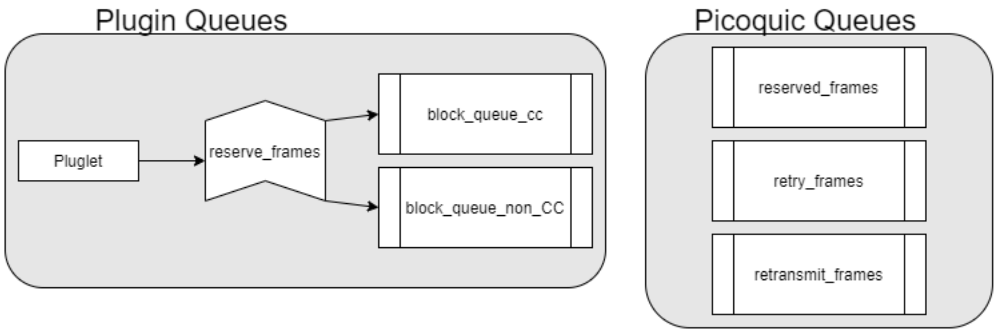


Figure 3.1: Visualization of how PQUIC implements Plugin and picoquic specific queues

3.3 PQUIC and Plugin Interaction

The PQUIC framework allows a plugin to introduce new frames with the restriction that they must have a unique and distinct frame identifier. Moreover, PQUIC provides an interface for pluglets to book frames for transmission by the picoquic connection. A picoquic connection manages three FIFO ordered queues, however, only one is related to plugin interaction (reserved_frames). Plugins maintain two separate FIFO ordered queues where one is dedicated to congestion-controlled traffic and one to non-congestion controlled traffic, see figure 3.1. A pluglet can book frames to be transmitted by picoquic through a common function shared by all plugins called *reserve_frames*. When a pluglet invokes *reserve_frames* the frame can be enqueued to either plugin queue depending on how it is marked. It is important to state that the reserved frames are **not** the frames that are transmitted by picoquic. The actual frames are written during packetization by a pluglet replaced protocol operation *write_frames* using the reserved plugin frames. When PQUIC prepares a packet for transmission it decides which frames to include in said packet. This is done by moving data from plugins queues to the picoquic *reserved_frames* queue. From this point, the picoquic is responsible for packetization and transmission.

The process of moving frames from plugins to picoquic is done through the protocol operation *schedule_frames_on_path* which uses a shared function *picoquic_frame_fair_reserve*. The function implements a two-layer DRR algorithm that cyclically alternates between the injected plugins. During the first pass, plugins with data in the congestion-controlled queue are considered and the second pass considers the non-congestion controlled plugin queues. Each function invocation can move a total amount of bytes equal to the space left in the current paths' congestion window. A plugin will move the maximum amount of data allowed and available in its queues before the next plugin is considered.

3.4 Datagram Plugin and PQUIC Virtual Private Network

The datagram plugin is based on a proposed datagram extension to the QUIC protocol[30]. The basic functionality is to allow for unreliable delivery of data while remaining ack-eliciting, i.e. acknowledging whether a datagram frame was received or not. The acknowledgments are used by the PQUIC congestion controller to adapt the transmission rate to the networks' current congestion state. The datagram plugin introduces new datagram specific frames and implements the required pluglets to support unreliable delivery of those frames. For an overview of which protocol operations are modified see appendix 4.

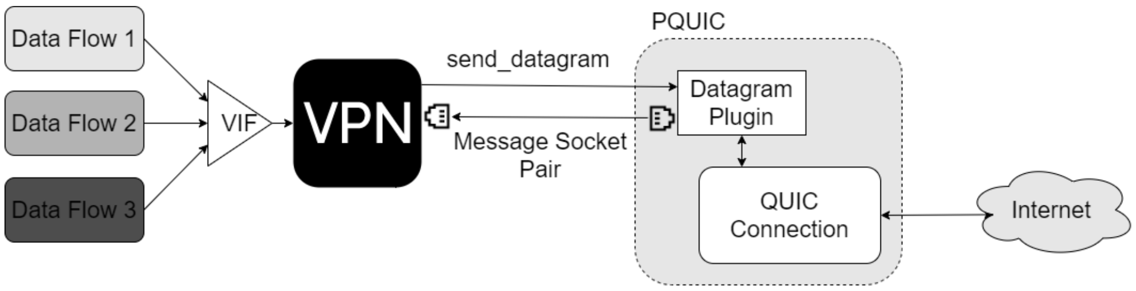


Figure 3.2: High level depiction of how the PQUIC datagram plugin operates within a picoquic connection

The design of the datagram plugin is to capture data from external applications and encapsulate it into the picoquic connection, see figure 3.2. This is done by implementing three *external* pluglets which can be invoked by a external applications.

- send_datagram
- get_datagram_socket
- get_max_datagram_size

The external application under consideration is a VPN implementation provided with the framework. The VPN is designed to capture traffic at IP level[5] through the use of virtual interfaces similar to the MPTCP and MPDCCP transport layer access bundling solution mentioned in section 2.3.1. This allows the mapping of any type of traffic, running any type of IP traffic over the VPN tunnel.

Sending Data Over PQUIC VPN When an IP datagram is available on the virtual interface the VPN invokes the *get_max_datagram_size* from the datagram plugin to determine the maximum allowed datagram size. This is equal to the maximum MTU considering all paths available from the picoquic connection. If the read datagram size is larger than the maximum allowed size by PQUIC the datagram is dropped by the VPN. Sequentially the datagram is passed on to the datagram plugin through the *send_datagram* external pluglet which encapsulates the IP datagram within PQUIC and reserves it for scheduling. When the datagram plugin reserves a frame it is marked as not restransmittable thus these frames cannot be retransmitted if lost. However, the picoquic connection maintains frame sequencing to support ack-eliciting.

Receiving Data Over PQUIC VPN When picoquic receives a packet containing one or more datagram frames, PQUIC can distinguish them due to the frame ID. Upon reaching a protocol operation anchor point, a datagram plugin-specific pluglet will be invoked. The datagram frame is parsed, de-encapsulated, and forwarded to the socket pair interconnecting the plugin with the VPN. Lastly, the VPN reads IP datagrams from the datagram socket exposed by the *get_datagram_socket* pluglet and forwards the data to the virtual interface.

3.5 Multipath plugin

PQUIC also provides a plugin to support multipath communication similar to MPQUIC, see section 2.2.2. The plugin adds support for path management, path migration, and packet scheduling by replacing protocol operation with pluglets related to sending and receiving picoquic packets. Due to the pluggable nature of PQUIC, the packet scheduler is implemented to be an interchangeable per-connection basis.

A multipath connection is initiated with a negotiation process where the client and server indicate that they are multipath compatible and which IP addresses they are accessible from. When the multipath connection is established the primary work is done by replacing the protocol operation *schedule_frames_on_path* which is used by all other plugins and the picoquic connection itself. It implements the path manager and integrates the packet scheduler to determine how the packets should be distributed over said paths. PQUIC is shipped with LowRTTand RR packet schedulers, discussed in section 2.2.3. The pluglet still utilizes the *picoquic_frame_fair_reserve* function to extract data from plugins thus only changing the behavior of the picoquic connection. PQUIC is still able to support multiple concurrent plugins to alter or add different functionalities. A multipath compatible receiver can distinguish between different paths through replacing the protocol operation *get_incoming_path*, used only by the picoquic connection.

4 Adapting FQ-CoDel to PQUIC

An important factor for FQ-CoDel to have any impact on network performance is if it's integrated where data flow multiplexing occurs. Due to the design of the tunneled transport layer access bundling and data flow encapsulation, this occurs at the transport layer. Therefore, the section starts with a discussion about the motivation for applying FQ-CoDel at the transport layer. This is followed by how the bottleneck queue of PQUIC was located and subsequent problems with the current single queue approach used by PQUIC. Following is a design section of how FQ-CoDel is implemented in PQUIC and the source code implementation with details surrounding it is presented. Finally, the section covers an explanation of what limitations the congestion controller in the PQUIC framework poses for implementation and evaluation and how this was resolved.

4.1 Motivation

Tunneled transport layer access bundling, see section 2.3.2, facilitates a generic solution to achieve multi-connectivity for UEs, such as smartphones, and can improve performance such as increased reliability and throughput. However, the introduction of tunneled transport layer access bundling alters the way data is designed to flow and behave when passing through the network stack. One core design is to support the multiplexing of data flows, at the transport layer, to a single MP connection thus creating a virtual tunnel between a client and host, see figure 2.5. The current approach for tunneled transport layer access bundling, see section 2.3.2, multiplexes data flows into a single FIFO ordered buffer in the VPN at the transport layer. The single queue can contribute to bufferbloat, see section 2.4, and problems with fair-queue access, see section 2.4.2, between active data flows. The multiplexing is done by capturing datagrams at IP level and appending them to the VPN buffer which probabilistically multiplexes the packets from different flows depending on the arrival order. The VPN encapsulates the different flows into a single MP connection by dequeuing data from the head of the buffer. An MP connection can utilize multiple network interfaces and thus multiple paths but the exposed IP 5-tuple from the MP connection will remain the same in the lower layers. Therefore, the multiplexing process removes the ability to distinguish between the different flows at lower layers of the network stack.

This mitigates performance optimization done at the link layer. The link-layer incorporates buffer management and a multiplexing process to distribute data flows over the physical link. Furthermore, by utilizing the IP 5-tuple the link layer can distinguish between different data flows and enforce fair buffer and link utilization. Existing solutions such as FQ-CoDel, described in section 2.4.3, aim to maintain a low queuing delay, and fair access to the physical media by separating application flows into different distinct buffers. Each buffer then operates using CoDel, see section 2.4, a AQM to mitigate bufferbloat and achieve a low queueing delay. By multiplexing dataflows into a single connection at the transport layer, as done by tunneled transport layer access bundling, revokes the link layer adaptation of FQ-CoDel to distinguish between active application flows. FQ-CoDel is thus only able to utilize a single CoDel queue and will suffer from fair-queue utilization mentioned in section 2.4.2.

For the tunneled transport layer access bundling to take advantage of queuing performance optimization, the functionality must be lifted to where the data flow multiplexing occurs. Thus, FQ-CoDel and other queue optimizations must be integrated into the tunneled transport layer access bundling and change the currently used FIFO queue. Integrating advanced queue functionality at the transport layer intends to provide an efficient multiplexing process and thus fair and access to the MP connection while maintaining a low flow queue delay. Furthermore, integration at the transport layer allows for the MP connections packet scheduler to take advantage of the multiplexing optimization.

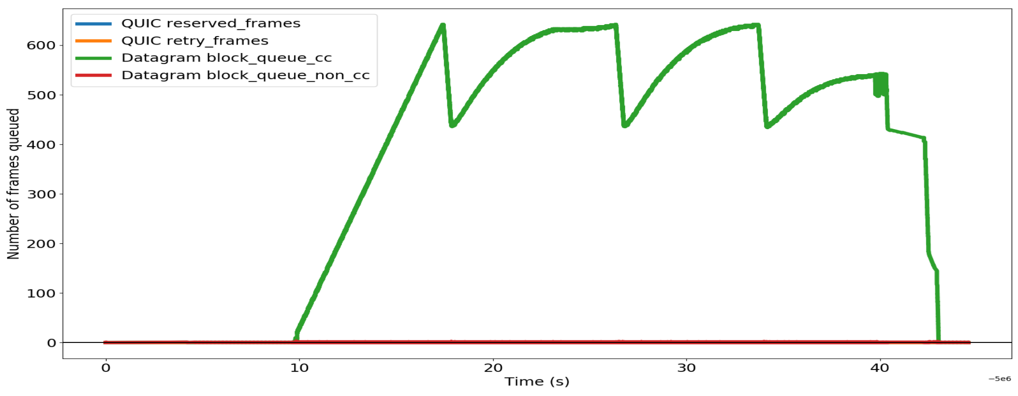
Table 4.1: Mininet network configuration for queue evaluation

Node Connection	**Bandwidth (Mbps)**	**Latency (ms)**	**Packet Loss (%)**
Router 1 - Router 3	20	10	0
Router 3 - Proxy	1000	0	0

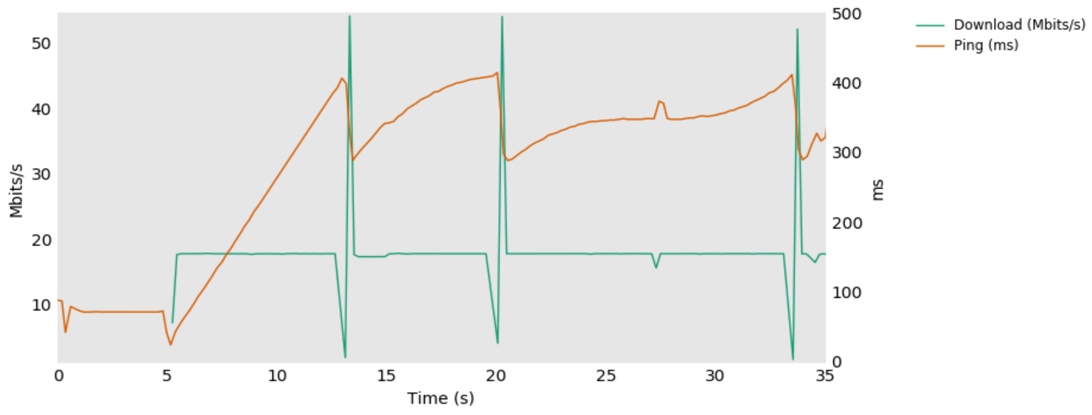
4.2 Considerations and Locating the PQUIC Bottleneck

For AQM to be effective for tunneled transport layer access bundling, motivated in section 4.1, the functionality must be integrated to buffers at the transport layer. Furthermore, due to the internal queue management of PQUIC, see figure 3.1, one must ensure where bufferbloat forms. This is evaluated using *Mininet*[31] to emulate a network topology which can be seen in figure 5.1. PQUIC runs the VPN application with only the datagram plugin inserted (creates a single path connection) and tracks all internal PQUIC queues, see figure 3.1 evolution over time. Furthermore, the client node runs the PQUIC VPN client and the proxy node runs the PQUIC VPN server. The network is configured according to table 4.1. All link-layer queueing disciplines run FQ-CoDel.

The test is executed by the client requesting a download from the *Web* node through the PQUIC VPN. This creates a 30-second long $\text{TCP}_{\text{Cubic}}$ downlink connection to measure throughput along with an Internet Control Message Protocol (ICMP) flow used to measure latency. A limitation is that PQUIC ensures that a packet is full before transmission (equal to the MTU of the physical link layer) and causes ICMP packets to be buffered before transmission and thus yielding varying RTT values. This is mitigated by enforcing an ICMP packet size equal to the MTU of the virtual interfaces resulting in immediate transmission.



(a) Queue Evolution over time in PQUIC



(b) $\text{TCP}_{\text{Cubic}}$ experienced throughput and latency

Figure 4.1: Test results for locating the bottleneck queue in PQUIC

The output from the test can be seen in figure 4.1. By observing the queue evolution over time in PQUIC it can be concluded that the bottleneck is located at the datagram plugin *block_queue_cc* queue. Therefore, FQ-CoDel must operate as a plugin-specific queue to ensure fair multiplexing and performance increases. Furthermore, observing the experienced throughput of $\text{TCP}_{\text{Cubic}}$ and the latency of ICMP the result indicates that the *block_queue_cc* queue contributes to bufferbloat, see section 2.4. The throughput remains constant at 20 Mbps and as the queue evolves the experienced latency increases.

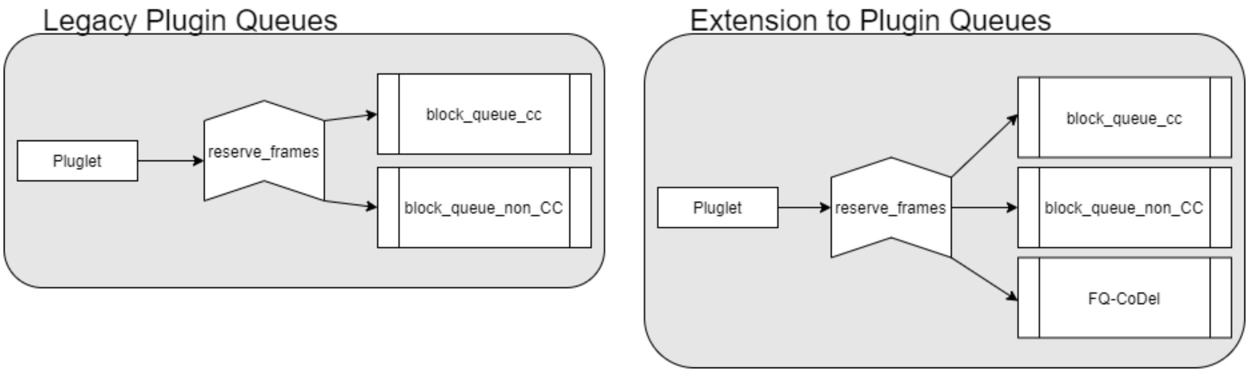


Figure 4.2: Visualizaion of how PQUIC plugins are extended to support FQ-CoDel

4.3 Design

The bottleneck queue is located at the datagram plugins' *block_queue_cc* queue and thus FQ-CoDel is integrated as a plugin-specific queuing discipline. To keep legacy functionality of frame reservation for pluglets we designed FQ-CoDel to use the legacy *reserve_frames* function and that the frame must be congestion controlled. Moreover, a plugin can opt-in and mark frames as FQ-CoDel compatible to reserve frames into FQ-CoDel, see figure 4.2. When a frame is reserved into FQ-CoDel it is time stamped and follows the procedure sequence as described in section 4.4.2 bellow. Utilizing the *reserve_frames* function requires only minor changes to source code and data structures to enqueue and dequeue frames from FQ-CoDel. Additions to the reserve frame structure include a flag that indicates FQ-CoDel compatibility and a *key* variable that stores the extracted IP 5-tuple from the incoming datagram. Additions to the dequeuing procedure from plugin-specific queues done by the *picoquic_frame_fair_reserve* function, see section 3.3, adds an additional layer to the DRR algorithm. This is done by checking for dequeuing opportunities in the FQ-CoDel queue first. If a frame can be dequeued from FQ-CoDel it follows the logic described in section 4.4.3 bellow.

The current implementation allows each plugin to manage a distinct instance of FQ-CoDel. However, due to the pluggable nature of PQUIC, it is possible to make FQ-CoDel into a plugin to only allow for PQUIC to manage a single FQ-CoDel instance. This will increase performance, save memory, and processing time if multiple plugins are heavily data-driven. For the scope of this thesis, the only data-heavy plugin is the datagram plugin, and as such plugganizing FQ-CoDel is left for future work.

Table 4.2: FQ-CoDel parameters

Number of FQ.CoDel Queues	1024
Memory Limit (MB)	10
CoDel Target Time (ms)	5
CoDel Interval Time (ms)	100
CoDel Credits (B)	1400

4.4 Implementation

The FQ-CoDel queuing discipline adaptation is based on the FQ-CoDel link-layer discipline in the Linux kernel presented by Dumazet[32] and CoDel presented by Nichols et al.[33]. FQ-CoDel is maintained by five main parameters seen in table 4.2 and the code is structured into three different parts; FQ flow classification, FQ enqueueing, FQ dequeuing. All queues maintained by FQ-CoDel are FIFO ordered. Dequeuing and dropping procedures are performed at the head of the queues.

4.4.1 Classification

An integral part of FQ-CoDel is being able to distinguish between different data flows. The PQUIC VPN captures raw IP packets and forwards them to the datagram plugin thus exposing the IP 5-tuple[4] accessible through the IP header. When a datagram frame is prepared for reservation to the FQ-CoDel queue discipline, the 5-tuple is extracted and stored as a key variable making it accessible during the FQ enqueueing procedure.

When a datagram frame is enqueued it is stochastically classified to an FQ-CoDel queue using the Jenkins hash function[34] as proposed by Request for Comments (RFC) 8290[7]. A stochastic approach poses low processing overhead but introduces problems with hash collisions as multiple flows can be classified into the same queue. When FQ-CoDel implements 1024 queues, as enforced by this implementation, the probability of a collision for 100 concurrent flows is equal to 90.78%[7] and is deemed sufficiently high for deployment.

4.4.2 Enqueuing

The enqueueing process, see algorithm 2, consists of classifying the incoming frame to an existing queue and timestamping it for CoDel. The timestamping allows the codel_dequeue procedure to measure a frames' sojourn time. If the queue was empty upon enqueueing the incoming frame, the queue is enqueued to the back of the new flows queue and gets credits equal to the quantum variable. The quantum is equal to the MTU defined by the PQUIC framework. The procedure is finalized by checking if FQ-CoDel has more data enqueued than the specified memory limit. If the limit is exceeded a frame will be dropped from the largest CoDel managed queue through a head drop.

[4]{source IP, destination IP, source port, destination port, and protocol}

Algorithm 2 FQ-CoDel enqueue function

```
1: procedure FQ_ENQUEUE(FRAME)
2:     queue_index = classify_flow(frame.key)        ▷ Classifies frame to a queue
3:     flow = flows[queue_index]
4:     codel_frame = {frame, current_time}           ▷ Timestamp frame for CoDel
5:     if flow_is_empty(flow) then
6:         enqueue(flow, new_flows)                  ▷ If flow is empty add it to new flows queue
7:         flow.credits = quantum                    ▷ add credits to flow
8:     enqueue(flow, codel_frame)                    ▷ Enqueue and handover to CoDel
9:     if Memory_limit_reached() then                ▷ Drop data from the fattest queue
10:        fq_codel_drop()
11:    return ok
```

4.4.3 Dequeueing

The procedure of dequeuing, see figure 3, prioritizes new flows over old flows through a two-layer DRR algorithm. If there exists a CoDel queue at the head of the queue with new flows and the chosen flow has credits available, a frame will be dequeued from the said flow. Old flows follow the same procedure. However, if the chosen flows' credits are exhausted, regardless if it is new or old, it will be moved to the back of old flows thus enforcing DRR. A frame is then dequeued by codel_dequeue. The CoDel implementation is based on the pseudo-code provided by Association for Computing Machinery (ACM)[35]. This is explained in section 2.4.1. A queue can be empty when CoDel tries to dequeue a frame, this will cause codel_dequeue to return nothing. If the empty flow was new it will be moved to the back of old flows and an empty old flow will be removed. This is a countermeasure to prevent starving old flows when new flows still have enough credits for dequeuing a frame but nothing to dequeue. Finally, the dequeued frame is returned to be handled by PQUIC.

Algorithm 3 FQ-CoDel dequeue implementing a 2-layer DRR

```
procedure FQ_Dequeue
    frame = NULL
    flow = NULL
    while !frame do
        if flow_not_empty(new_flows) then
            flow = queue_head(new_flows)
        else if flow_not_empty(old_flows) then
            flow = queue_head(old_flows)
        else
            return
        if flow.credit < 0 then
            flow.credit += quantum
            move_queue(flow, old_flows)
            continue                                    ▷ Consider next queue
        frame= codel_dequeue(flow)                      ▷ Returns NULL if dequeue fails
        if !frame then
            if is_new_flow(flow) & flow_not_empty(old_flows) then
                move_queue(flow, old_flows)             ▷ Avoid starving old flows
            else
                delete_flow(flow)                       ▷ CoDel emptied queue so it must be removed
    return pkt
```

4.5 PQUIC Congestion Controller Limitations

The current implementation of PQUIC provides a pluggable congestion controller with support for Cubic [36] and New Reno [37]. The evaluation and effect of different internal congestion controllers are outside the scope of this thesis and as such Cubic will be used for all evaluations.

Another problem related to the congestion controller is the way it interacts with the datagram plugin. All datagram plugin frames are marked as *non-retransmittable* which causes bad interaction with how PQUIC integrates checks for congestion control functionality. All congestion control backoff functionality, e.g. frame loss or fast retransmit, takes place during procedures only affecting retransmittable frames. The datagram frames do not enter these procedures and as such cannot ever reduce the congestion window. However, the frames are still acknowledgment-eliciting and congestion window increases are possible. For a long-lived connection, the congestion window will grow infinitely manifesting itself as a non-congestion controlled connection. This is mitigated by enforcing a static congestion window equal to the Bandwidth-delay Product (BDP) for a given path. The connection will thus be constrained by the static congestion window while remaining congestion-controlled through acknowledgment-eliciting. This is enforced for all testing and evaluations.

$$BDP = Bandwidth(bit/s) * RTT(s) \tag{4.1}$$

5 Test Design

The following section introduces the testing environment that is used for evaluation and the different network configurations that are applied. Following this is an explanation of how PQUIC is configured to work with as intended in conjunction with the network configurations. Finally, the test cases, used for evaluation are presented.

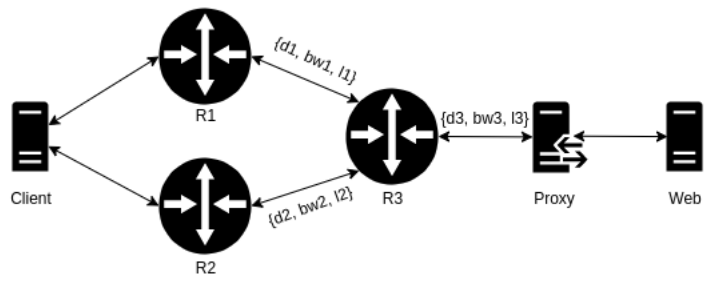


Figure 5.1: Visualizaion of the configured mininet topology

5.1 Test Environment

To evaluate the performance of the interaction between the multiplexing of FQ-CoDel and the packet schedulers of PQUIC there is a need for a test environment. The test environment intends to facilitate a network topology that supports multi-connectivity between end nodes, see figure 5.1. The network topology consists of the client node which supports multi-homing and hosts the PQUIC VPN client with the datagram and multipath plugin injected. Furthermore, the proxy node hosts the PQUIC VPN server with the datagram and multipath plugin injected. Router 1 and Router 2 forms the disjoint networks and Router 3 acts as the aggregation point for said networks. To establish the network the Mininet[31] network emulator is used.

Even though network emulation can introduce unknown variables that can impact the result it is chosen due to congestion controller limitations posed by PQUIC, see section 4.5. The current state of PQUIC enforces a static per-path congestion window calculated from the BDP given by the network topology. A similar topology can be implemented as a private LAN but proves inefficient for evaluation due to a high amount of reconfiguration of network settings and changes in source code between tests.

5.2 Network And PQUIC Configuration

The network topology is configured to account for two different network scenarios where homogeneous network paths and heterogeneous network paths are considered. For both configurations the maximum allowed bandwidth for a given bottleneck link is set to 20 Mbps. Higher bandwidth causes PQUIC to maximize processor utilization and introduces performance degradation as a result. De Coninck et al. [5] presents that the current state of PQUIC is not performance-optimized but focuses instead of providing good extendability through plugins.

Table 5.1: Configuration for homogeneous Network, all unnamed links are configured as Router 3 - Proxy

Node Connection	**Bandwidth (Mbps)**	**Latency (ms)**	**Packet Loss (%)**
Router 1 - Router 3	20	10	0
Router 2 - Router 3	20	10	0
Router 3 - Proxy	1000	0	0

Table 5.2: PQUIC path configuration for homogeneous network evaluation

Path Name	**Network Link Connection**	**Static Congestion Window (B)**
Path A	(Router 1 - Router 3)	50000
Path B	(Router 2 - Router 3)	50000

The configuration for a homogeneous network can be seen in table 5.1 and the corresponding PQUIC path configuration in table 5.2. The static per-path congestion window for PQUIC corresponds to the network configuration. As mentioned before, these values are derived from the BDP from the corresponding link.

Table 5.3: Configuration for heterogeneous Network, all unnamed links are configured as Router 3 - Proxy

Node Connection	**Bandwidth (Mbps)**	**Latency (ms)**	**Packet Loss (%)**
Router 1 - Router 3	20	5	0
Router 2 - Router 3	10	10	0
Router 3 - Proxy	1000	0	0

Table 5.4: PQUIC path configuration for heterogeneous network evaluation

Path Name	**Network Link Connection**	**Static Congestion Window (B)**
Path A	(Router 1 - Router 3)	25000
Path B	(Router 2 - Router 3)	25000

The configuration for a heterogeneous network can be seen in table 5.3 and the corresponding PQUIC path configuration in table 5.2. The emulated approach poses one link (path a) with significantly better better characteristing similar to UE with good access to LTE and a deteriorating connection to WLAN.

5.3 Test Methodology

The testing procedure covers a set of scenarios that tunnels different types and amounts of traffic through the PQUIC VPN. Each scenario and their purpose are described in table 5.5. The scenarios are evaluated using combinations of PQUIC legacy single queue and FQ-CoDel. Further variables are the different PQUIC packet schedulers, RR, and LowRTT, and heterogeneous and homogeneous network configuration. The tests are executed using The FLExible Network Tester (FLENT) [38] a wrapper for well-known benchmarking tools such as iperf. FLENT allows for repeatable tests and enables the use of multiple concurrent dataflows.

Table 5.5: Test design and purposes

Test Name	Design	Purpose
Single TCP_{cubic} and ICMP	Single TCP downlink with concurrent ICMP	Evaluating behaviour of FQ-CoDel and compare against legacy single FIFO queue
Two Delayed TCP_{cubic}	Single TCP downlink with a second TCP downlink initiated after 5s	Evaluating fair access to medium and convergence speed
Four TCP_{cubic} and ICMP	Four concurrent TCP downlink with a concurrent ICMP downlink	Evaluating FQ-CoDel bufferbloat mitigation and fair multiplexing between flows
Realtime Response Under Load	Four concurrent TCP downlink and four concurrent TCP uplink with a concurrent ICMP downlink	Evaluation of system under high stress

6 Results and Evaluation

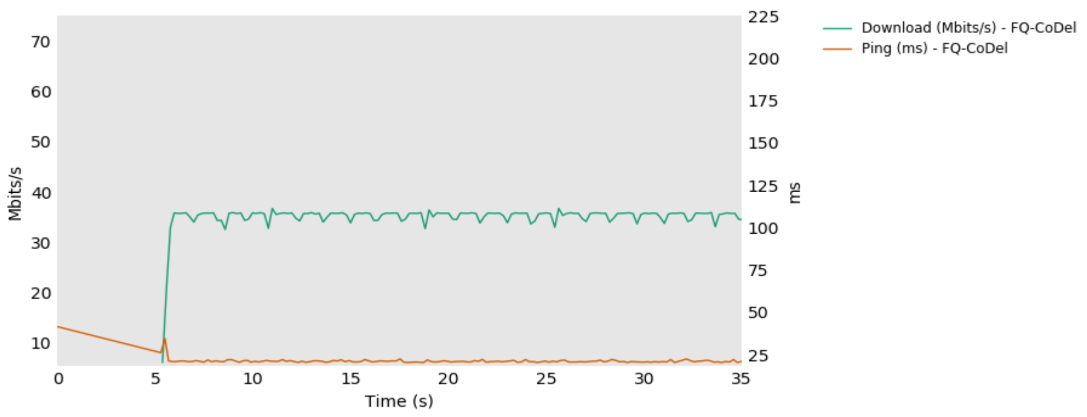
The following section presents the results from the different tests when different types and amount of traffic is tunneled over the PQUIC VPN. Following is a discussion about the problem the current implementation of PQUIC presents for real-time traffic with a continuous sending pattern. The section is finalized by an evaluation of the results.

6.1 Single TCP Flow

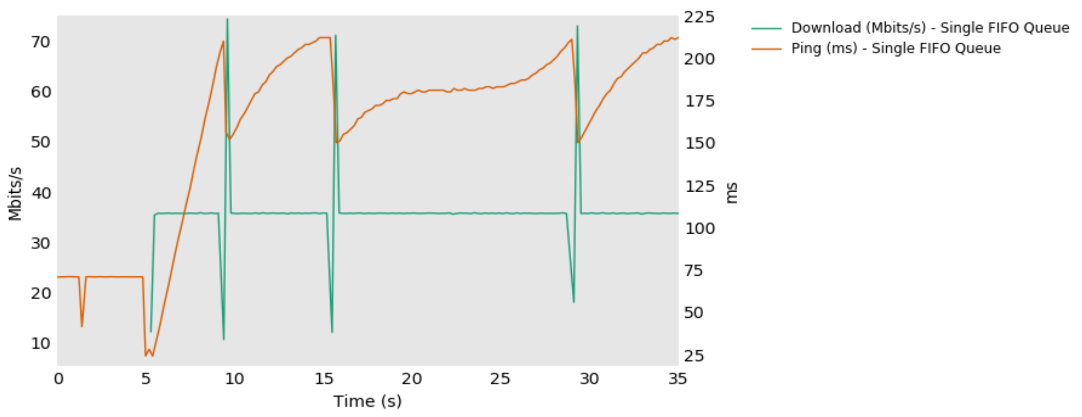
Figure 6.1 visualizes a downlink TCP and ICMP flow tunneled over the PQUIC VPN. It compares FQ-CoDel as a transport layer queueing algorithm and the legacy PQUIC single FIFO queue. The test is executed in the homogeneous network topology, see section 5.2 and the RR packet scheduler is used.

The RR scheduler distributes packets over the available paths on a per-packet basis and is only able to be blocked by the static per-path congestion window. Given the homogeneous network, the RR packet scheduler distributes the data 50,1% over path A and 49,9% over path B. Figure 6.1b further shows that the RR scheduler aggregates the capacities of the network paths and can achieve close to perfect resource pooling. Even though a total of 40 Mbps is available when the network paths are aggregated, PQUIC is only able to utilize an average of 36 Mbps. This discrepancy is due to, overhead generated by the encapsulation process, and lack of performance optimization of PQUIC.

Figure 6.1b further indicates that FIFO queue adaptation suffers from bufferbloat due to the high latency experienced by the ICMP flow. The dips in throughput are caused by an overfull sending queue in the VPN, resulting in frame drops which causes the TCP flow to reduce its congestion window. The frames are dropped at the head of the queue to provide better responsiveness for tunneled application flows. The subsequent peak is caused by buffering incoming packet, as PQUIC is in a dropping state incoming packets are buffered which is manifested as a temporary increase in throughput.



(a) Experienced throughput and latency for FQ-CoDel



(b) Experienced throughput and latency for single FIFO Queue

Figure 6.1: Results from 1 TCP downlink flow with a concurrent downlink ICMP flow, comparing transport layer adaptation of FQ-CoDel and single FIFO queue while using the PQUIC RR packet scheduler over a homogeneous network

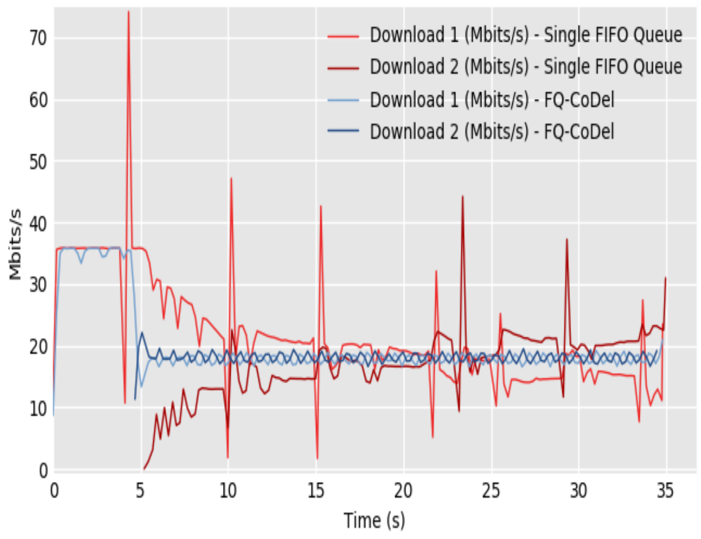
The adaptation of FQ-Codel is configured according to table 4.2 with a target setting of 5ms and an interval setting of 100ms. Thus if an FQ-CoDel queue has a sojourn time surpassing the target time for at least an interval FQ-Codel enters a dropping state. Figure B.1b in appendix B depicts the FQ-CoDel queue evolution over time and the experienced sojourn for frames overtime for the first 8 CoDel queues. When FQ-CoDel enters a dropping state for a given queue, frames from that queue are continuously dropped until the sojourn time is below the target limit or the queue is empty. These drops are manifested as a reduction of the affected TCP flows congestion window and explains the throughput dips experienced using FQ-CoDel in figure 6.1a. Furthermore, by enforcing flow queueing and a low queuing delay a low experienced latency is achieved as the flows have fair access to the picoquic connection. Due to the use of the RR packet scheduler and a homogenous network topology the experienced latency for the ICMP flow averages 24ms of 20ms possible. This is also depicted in figure 6.1a and in appendix B.1a. The discrepancy is expected due to the encapsulation and decapsulation process, and additional overhead is generated by PQUIC.

6.2 Two Delayed TCP Flows

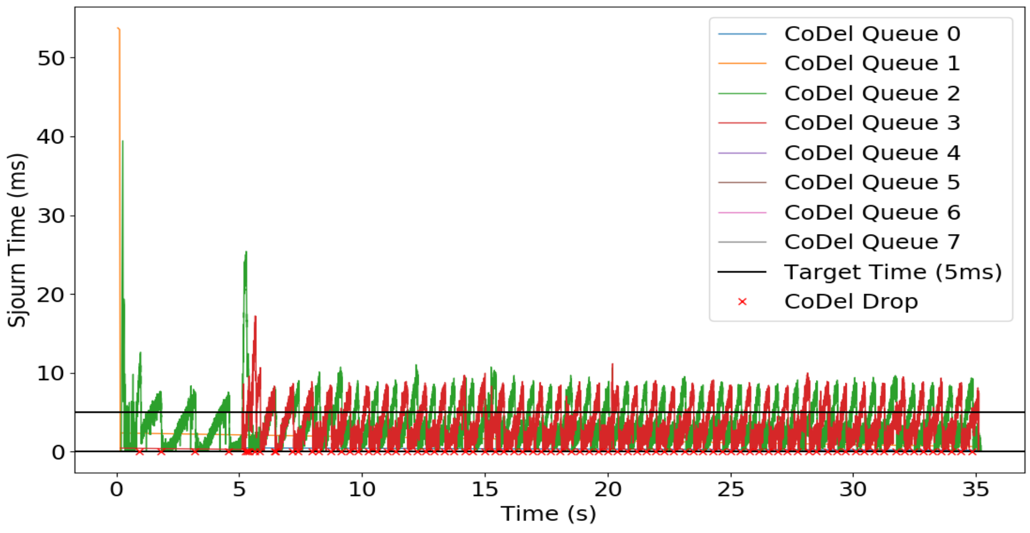
Figure 6.2 visualizes two downlink TCP flows tunneled over the PQUIC VPN where one flow is delayed 5s. It compares how FQ-CoDel as a transport layer queueing algorithm and the legacy single FIFO queue copes with the introduction of new flows and concurrent flows. The test is executed in the homogenous network topology, see section 5.2, and the RR packet scheduler is used.

The single FIFO queue presents a problem with fair-queue utilization when the second TCP flow is introduced. When the second flow is introduced the first flow has formed a substantial queue inside PQUIC FIFO to what can be seen in figure 4.1a. When the buffer eventually overflows, frames are dropped from the head of the queue. The drops occur probabilistically depending on the current distribution of packets from different flows occupying the queue. When a frame is dropped, the corresponding flow will reduce its congestion window. This is manifested as a slow convergence to fair picoquic connection utilization over 5s. Due to the probabilistically dropping approach, the flows struggles to maintain a steady fair utilization. The dips and peaks follow the same logic as described in section 6.1.

Figure 6.2: Results from 2 TCP downlink flows where 1 flow is delayed 5s, comparing transport layer adaptation of FQ-CoDel and single FIFO queue using the PQUIC RR packet scheduler over a homogeneous network



(a) Experienced throughput comparing FQ-CoDel and single FIFO Queue



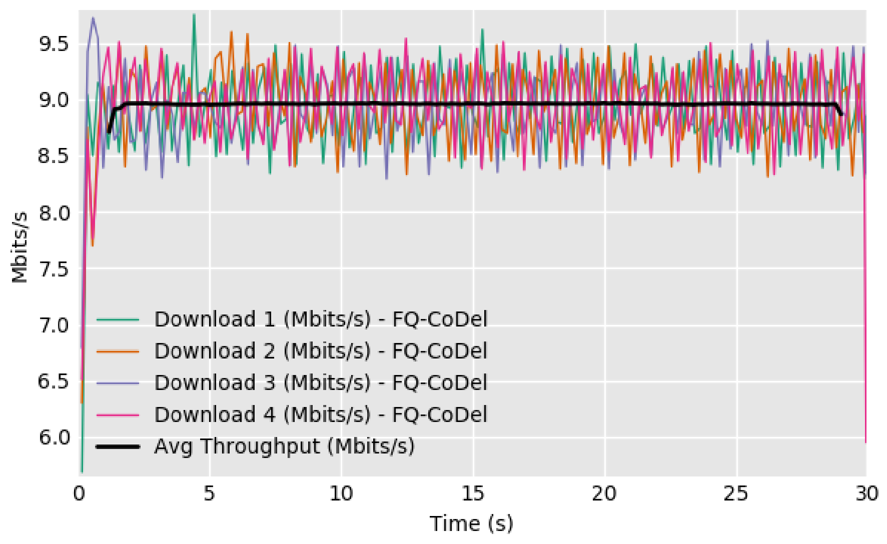
(b) FQ-CoDel per-queue sojourn evolution over time

When FQ-CoDel is applied as a queueing discipline the different TCP flows are stochastically classified to different CoDel queues. The dequeuing procedure integrates a DRR approach that cyclically iterates between the available CoDel queues while prioritizing new flows. The second flow thus has instant access to the picoquic connection and improves the first response delay, see figure 6.2a. Furthermore, the DRR adaptation allows for instant and continuous fair access to the underlying connection. In figure 6.2b the first 8 flows and CoDel queues are represented. The first TCP flow is represented as Flow 3 and the second TCP flow as Flow 1, red crosses mark any CoDel drop. The peak in queued bytes and sojourn time is caused by the slow start functionality of $\text{TCP}_{\text{Cubic}}$. After a CoDel interval elapses, frames from the bloated queue are dropped until a dequeued frame has a sojourn time smaller than the set CoDel target. The drops cause the TCP flows to enter the congestion avoidance state thus reducing the size of the congestion window and the increased rate of its congestion window. This allows for an FQ-CoDel queue to converge to the CoDel target limit within two CoDel intervals and maintaining a low queue delay. The congestion window increase functionality of TCPs is manifested as a constantly growing CoDel queue and subsequent drops when the sojourn time has surpassed the CoDel target for a total of a CoDel interval.

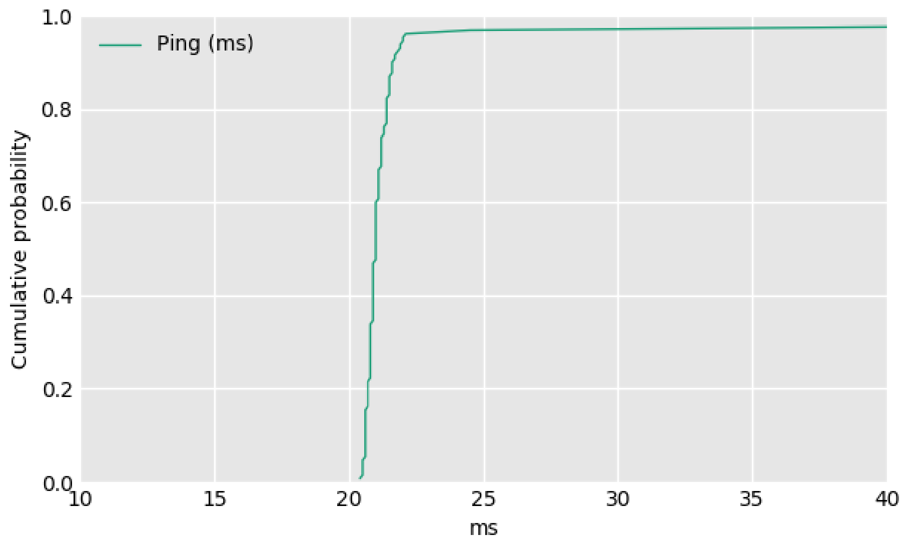
6.3 Four Concurrent TCP Flows

Figure 6.3 visualizes four concurrent downlink TCP flows and one downlink ICMP flow tunneled over the PQUIC VPN. It represents how FQ-CoDel as a transport layer queueing discipline copes with multiple concurrent flows and how this impacts latency. The test is executed in the homogenous network topology, see section 5.2, and the RR packet scheduler is used.

Figure 6.3: Results from 4 TCP downlink flows and 1 concurrent downlink ICMP flow, applying transport layer FQ-CoDel and evaluating the PQUIC RR over a homogeneous network



(a) Experienced throughput

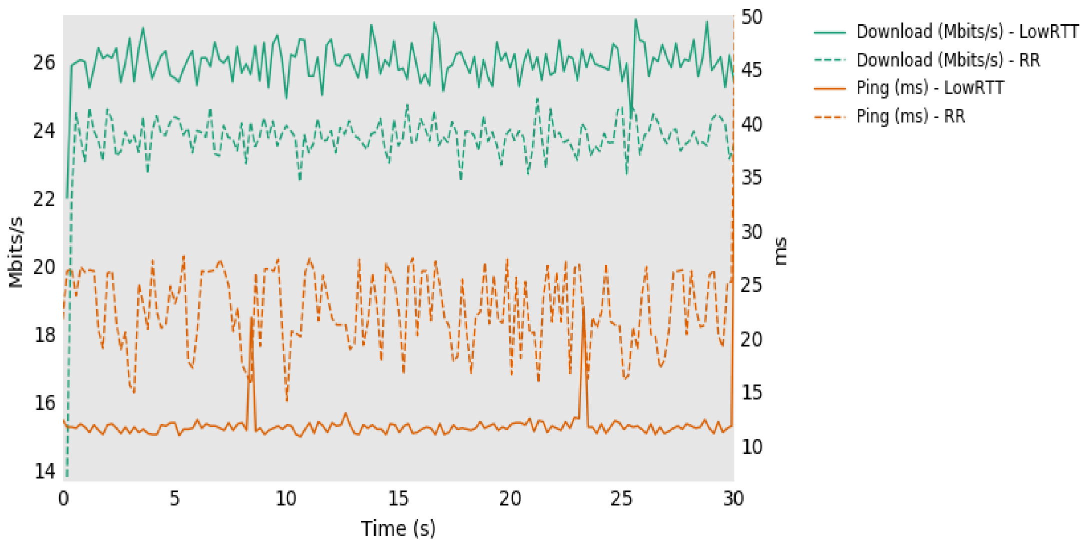


(b) Cumulative distribution of latency using 4 concurrent TCP downlink flows

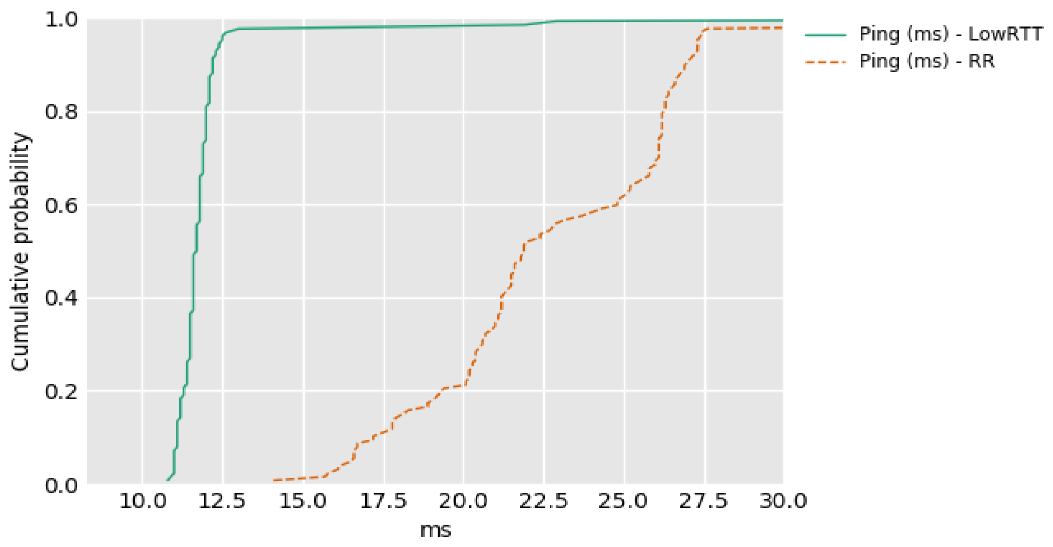
FQ-CoDel stochastically classifies all active application flows to separate CoDel queues which enforce flow queueing. The dequeuing procedures' DRR algorithm allows for CoDel queues to have fair access to the picoquic connection. Figure 6.3a and 6.3b verifies that fair access to the underlying picoquic connection is guaranteed and a low queuing delay can be enforced. The average throughput for a given flow is 9 Mbps which aggregated equals 36 Mbps, equivalent to the utilization of a single TCP flow while maintaining an average latency of 24ms.

Further evaluation of comparison between the RR, and LowRTT packet scheduler using 4 concurrent downlink TCP flows can be seen in appendix B.4. Due to the symmetric nature of the network, the behavior of the different packet schedulers is, as expected, indifferent. The RR scheduler fairly distributes packets over the available network on a per-packet basis and archives a packet distribution of 50,1% over path A and 49,9% over path B. The LowRTT scheduler prioritizes the path with the lowest measured RTT and when the congestion window is filled data is transmitted over the path with the second-highest measured RTT. Given a long-lived connection and a homogenous network, the LowRTT becomes ack-clocked similar to RR and has an equivalent packet distribution. The behavior of FQ-CoDel remains the same regardless of the packet scheduler used. Fair access to the underlying connection is guaranteed between the different application flows while a low queueing latency is enforced. The figure further indicates that the by adaption AQM in tunneled transport layer access bundling solution can mitigate bufferbloat that occurs at the transport layer.

Figure 6.4: Results from 4 TCP downlink flows and 1 concurrent downlink ICMP flow, applying transport layer FQ-CoDel while comparing the PQUIC RR packet scheduler and PQUIC LowRTT packet scheduler over a heterogeneous network



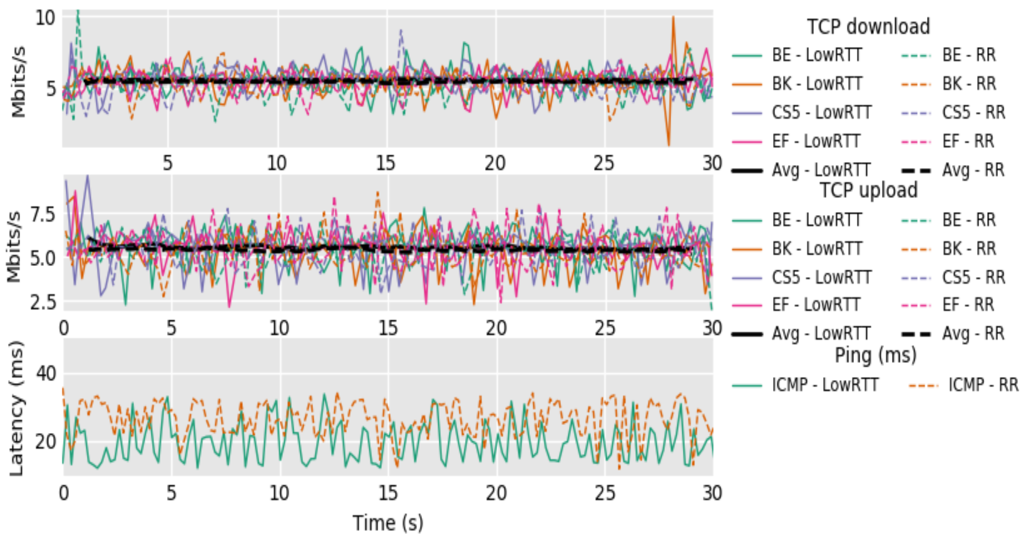
(a) Experienced throughput comparing RR and LowRTT packet scheduling



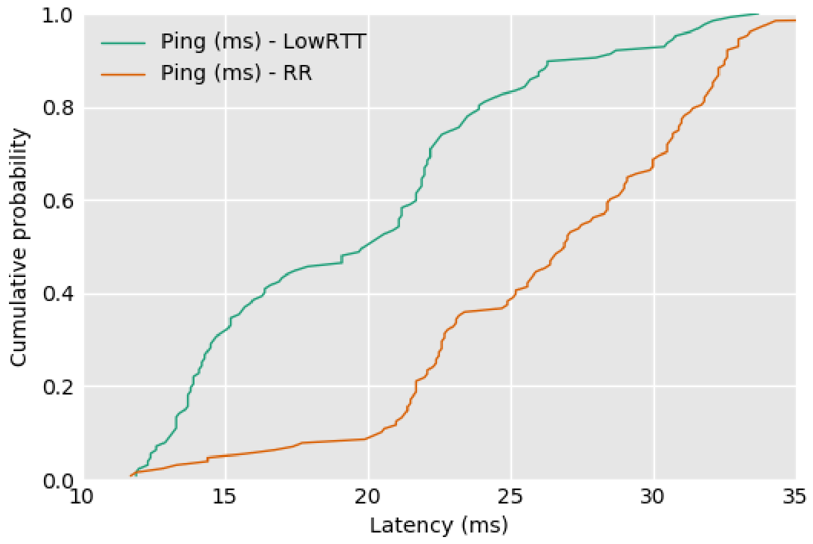
(b) Cumulative distribution of latency comparing RR and LowRTT packet scheduling

Figure 6.4 extends the evaluation by comparing the interaction between FQ-CoDel and the different packet schedulers in PQUIC over a heterogeneous network, see table 5.3. FQ-CoDel continuous to maintain fair access to the underlying connection between the different application flows while allowing for the packet scheduler module to distribute the packets over the available paths. The RR scheduler fairly distributes the packets over the available paths on a per-packet basis and due to the heterogeneous nature of the network and the varying link latencies causes out-of-order delivery at the receiver. The RR scheduler sends 62% of the traffic over the fastest path with the lowest RTT value, path A, but enforces cyclic iteration between the available paths when the paths are non-congested. The impact is a reduction in end-to-end application flow throughput due to required reordering at the receiver and increased experienced latency due to the fair packet distribution. The aggregated throughput average is 24 Mbps out of a total of 30 Mbps available and the distribution of latency is 28 ms compared to the theoretical minimum of 20 ms. Some discrepancy is expected as mentioned in section 6.1. The LowRTT packet scheduler prioritizes the path with the lowest measured RTT and is thus able to distribute the packets accordingly. The LowRTT scheduler distributes 68% of the traffic over the best path and always prioritizes it if non-congested. By prioritizing the path with the lowest RTT the amount of reordering at the receiver side is reduced and thus a higher throughput can be achieved. Further performance improvements by path prioritization is a reduced experienced latency by 54% compared to the RR scheduler.

Figure 6.5: Results from 4 TCP downlink flows, 4 concurrent TCP uplink flows and 1 concurrent downlink ICMP flow, applying transport layer FQ-CoDel while comparing the PQUIC RR packet scheduler and PQUIC LowRTT packet scheduler over a heterogeneous network



(a) Experienced throughput, both downlink and uplink, comparing RR and LowRTT packet scheduling



(b) Cumulative distribution of latency comparing RR and LowRTT packet scheduling

6.4 Real-time Response Under Load

Figure 6.3 visualizes four concurrent downlink TCP flows, four concurrent uplink TCP flows, and one downlink ICMP flow tunneled over the PQUIC VPN. It presents how FQ-CoDel as a transport layer queueing algorithm copes with multiple concurrent flows and how this impacts latency. The test is executed in the heterogeneous network topology, see section 5.2 and the RR and LowRTT packet schedulers are compared. The test is designed to evaluate how PQUIC and FQ-CoDel perform under high load. Figure 6.5a indicates that even under high-stress FQ-CoDel can improve the end-to-end performance for application flows tunneled through the PQUIC VPN and the perceived throughput distribution between the application flows remains fair when FQ-CoDel flow queueing is applied. Figure 6.5b further confirms that the experienced latency can be reduced if FQ-CoDel is combined with a packet scheduler that accounts for RTT in the network.

A comparison of how the single FIFO queue behaves with multiple concurrent flows can be seen in appendix B.5. The tests evaluate the RR scheduler over a homogeneous network and the LowRTT scheduler over a heterogeneous network. FQ-CoDel outperformance the single FIFO queue in regards to end-to-end latency and fair utilization in all cases.

6.5 PQUIC VPN Problem

Figure 6.6 visualizes one concurrent downlink Self-Clocked Rate Adaptation for Multimedia (SCReAM) flow tunneled over the PQUIC VPN. It presents how the PQUIC framework copes with real-time traffic with a continuous sending pattern. The test is executed in the homogeneous network topology, see section 5.2 and FQ-CoDel is applied. SCReAM is a congestion-controlled transport layer that targets real-time data transmission and generates traffic that does not enforce reliable delivery while remaining congestion-controlled through ack-eliciting developed by Ericsson [39]. Ericsson further provides a benchmarking application that can be executed in the mininet emulated network and allow for traffic tunneling over the PQUIC VPN. The interaction between the real-time traffic and the PQUIC VPN yields odd results. As seen in figure 6.6 a maximum throughput of 2 Mbps is perceived by the SCReAM application flow when it is tunneled through the PQUIC VPN. When SCReAM operates outside the VPN tunnel the maximum available bandwidth is utilized. Similar behavior is observed when a single UDP flow is tunneled through the VPN and when the single FIFO queue is applied. Observing the queue evolution inside FQ-CoDel and the sojourn time for queued frames there are few queued bytes while the sojourn time is tangent to the target time, see appendix B.7. The low throughput with a high queuing delay is most likely an effect of the design of the VPN. SCReAm generates a constant data stream that causes backpressure at the PQUIC VPN and forces it to constantly context switch between reading and transmitting data. The VPN is a single thread application and the context switching causes overhead which is manifested as CoDel sojourn time increase and reduced perceived throughput by SCReAM. The reasoning why TCP does not present similar problems is due to its bursty nature compared to the constant dataflow posed by real-time applications. TCP does not generate a constant pressure at the VPN which allows for context switching to occur without significant impact on the throughput and queueing delay.

The purpose of a tunneled transport layer access bundling solution that supports unreliable delivery of data is to tunnel any IP-layer traffic. Therefore, this should be investigated further and is of high priority for future work.

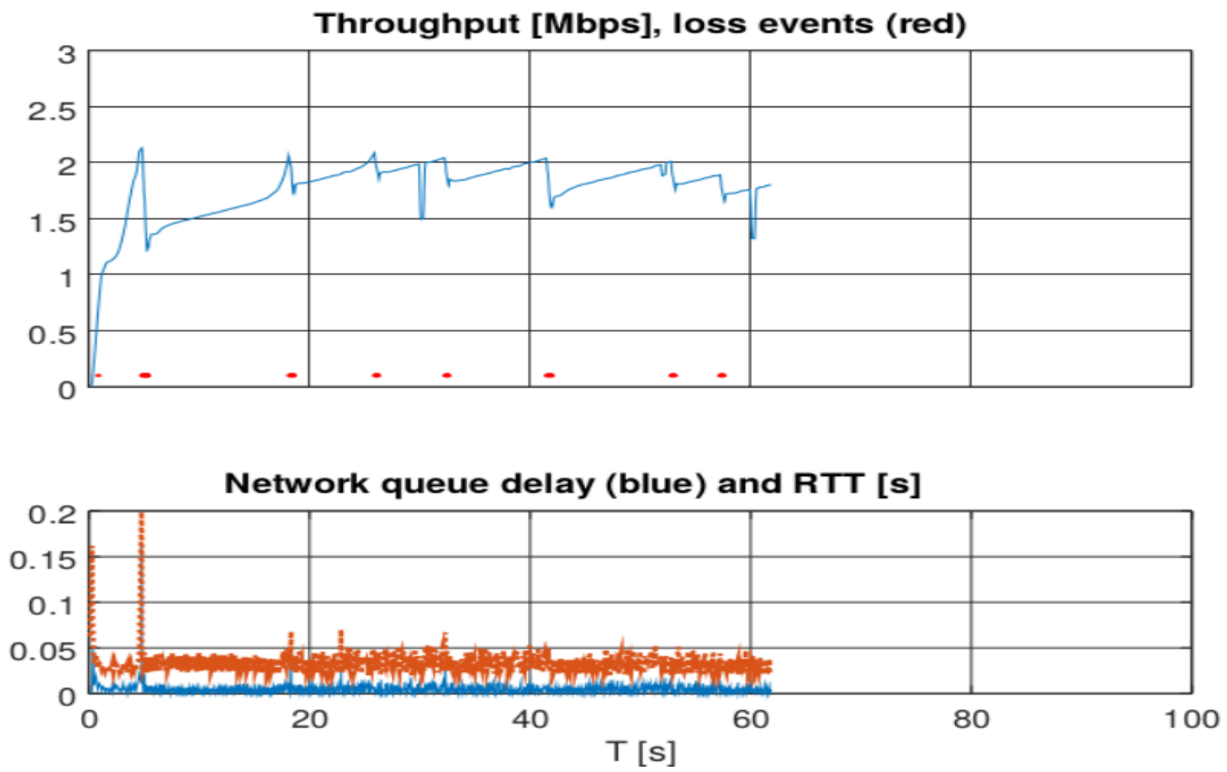


Figure 6.6: Results from 1 SCReAM downlink flow, applying transport layer FQ-CoDel and the PQUIC RR packet over a homogeneous network

6.6 Evaluation

The *Single TCP Flow* test indicates that it is possible to apply FQ-CoDel to tunneled transport layer access bundling solutions. It improves the latency compared to a single FIFO queue by enforcing per-flow queueing to allow the TCP and ICMP to occupy separate queues while enforcing a low queuing delay by applying CoDel AQM per-queue. The cumulative distribution difference in end-to-end latency between the single FIFO queue and FQ-CoDel indicates a percentage decrease of 88% of experienced latency when FQ-CoDel is applied. The *Two Delayed TCP Flow* test proves that FQ-CoDel can achieve close to instant fair share convergence to the underlying connection between competing flows and that and that this is maintained over the flows lifetime. The single FIFO queue poses problems with both slow fair share convergence and fair share maintenance due to the probabilistic approach to filling and dropping from the queue. The *Four Concurrent TCP Flows* test evaluates the impact between different packet schedulers and FQ-CoDel. Again, FQ-CoDel maintains fair access to the underlying connection between application flows. The packet schedulers yield performance differences over the heterogeneous network configuration where the LowRTT scheduler outperforms the RR scheduler in every test. The cumulative distribution of latency between the RR and the LowRTT scheduler indicates a reduction in end-to-end latency by 54% when using the LowRTT scheduler. Furthermore, the LowRTT scheduler improves the average end-to-end throughput by 7,5%. Test *Real-time Response Under Load* further indicates end-to-end performance improvements for application flows using the LowRTT scheduler over heterogeneous networks. It is thus possible to conclude that the interaction between FQ-CoDel as a queueing discipline for tunneled transport layer access bundling solutions in conjunction with packet schedulers can improve the performance when multiple application flows are tunneled concurrently. This presents a good foundation for further research.

7 Conclusion and Future Work

This thesis investigated if the end-to-end performance for application flows could be improved in tunneled access bundling solutions. For evaluation purposes, the PQUIC framework, an adaptation of the draft-14 QUIC protocol, was utilized. The framework supports multipath communication and unreliable delivery of data which allows for any type of IP-traffic to be tunneled over the VPN in a timely fashion. The common adaptation of a single FIFO queue, used for buffering data that is to be transmitted over the VPN, presents problems with fair share multiplexing between application flows and is susceptible to bufferbloat. This was evaluated and proven to degrade the end-to-end performance for application flows in regards to latency and fair throughput distribution, i.e. access to the VPN. Inspired by previous work on queuing disciplines at the link layer that enforces low queuing delay and fair access to the physical link, we applied a transport layer adaptation of FQ-CoDel. In our approach, we add support for fair application flow multiplexing while also mandating a low queuing latency, all performed at the transport layer. The evaluation shows that the FQ-CoDel can grant instant, constant and fair access to the VPN while significantly lower the end-to-end latency for tunneled application flows. Furthermore, applying a packet scheduler that adapts to current network characteristics, such as LowRTT, upholds the performance over heterogeneous networks while keeping the benefits of FQ-CoDel.

The work presented in this thesis does not pose any ethical problems but presents social impacts. Tunneled transport layer access bundling and the adaptation of FQ-CoDel opens up for new and improved network services for UEs in rural areas which lack support for high bandwidth access networks such as fiber to the home. Such a service is access network aggregation to support higher available bandwidths for end-users and better connection reliability by e.g. concurrently utilizing cellular networks and Digital Subscriber Line (DSL). Access network aggregation in combination with path scheduling can improve the quality of service for low-latency applications such as real-time multimedia streaming, by prioritizing network paths with lower RTT values. FQ-CoDel further extends this by ensuring resource fairness between competing application flows and mitigating excessive queuing delays. The solution presented in this thesis, therefore, allows tunneled transport layer access bundling to increase the quality of service for application flows by enforcing responsiveness through low queuing delays and fair resource utilization between said flows.

Propositions for future work is based on the design decisions and problems encountered during the development and the acquired results, the following should be considered.

- Plugganize the FQ-CoDel adaptation in PQUIC to improve the VPN performance.
- Locate and fix the problem with PQUIC VPN related to the constant flow of traffic and congestion controller problems within the datagram plugin.

- Extend the evaluation to evaluate a wider set of networks and further test in a non-closed environment.
- Experiment with different queuing disciplines adapted at the transport layer and different packet schedulers.

References

[1] Sandvine. The global internet phenomena report october 2018, oct 2018. `https://www.sandvine.com/hubfs/downloads/phenomena/2018-phenomena-report.pdf`, last accessed on 2020-05-24.

[2] 3GPP. Study on access traffic steering, switch and splitting support in the 5g system architecture(r16). TR 23793, dec.

[3] Xiaolan Liu, Danfeng Shan, Ran Shu, and Tong Zhang. Mptcp tunnel: an architecture for aggregating bandwidth of heterogeneous access networks. *Wireless Communications and Mobile Computing*, 2018, 2018.

[4] Markus Amend, Eckard Bogenfeld, Anna Brunstrom, Andreas Kassler, and Veselin Rakocevic. A multipath framework for UDP traffic over heterogeneous access networks. Internet-Draft draft-amend-tsvwg-multipath-framework-mpdccp-01, Internet Engineering Task Force, July 2019. Work in Progress.

[5] 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*, pages 59–74. 2019.

[6] Adam Langley, Al Riddoch, Alyssa Wilk, Antonio Vicente, Charles 'Buck' Krasic, Cherie Shi, Dan Zhang, Fan Yang, Feodor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Christopher Dorfman, Jim Roskind, Joanna Kulik, Patrik Göran Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, and Wan-Teh Chang. The quic transport protocol: Design and internet-scale deployment. 2017.

[7] Toke Hoeiland-Joergensen, Paul McKenney, dave.taht@gmail.com, Jim Gettys, and Eric Dumazet. The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm. RFC 8290, January 2018.

[8] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark buffers in the internet. *Queue*, 9(11):40–54, 2011.

[9] Felix Andersson. Pquic fq-codel extension. `https://github.com/FajFs/pquic`, last accessed on 2020-03-09.

[10] James F Kurose. *Computer networking: A top-down approach featuring the internet, 3E*. Pearson Education India, 2005.

[11] Jon Postel. Internet protocol. STD 5, September 1981.

[12] J. Postel. User datagram protocol. STD 6, August 1980.

[13] Jon Postel. Transmission control protocol. STD 7, September 1981.

[14] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. Internet-Draft draft-ietf-quic-transport-27, Internet Engineering Task Force, February 2020. Work in Progress.

[15] Giorgos Papastergiou, Gorry Fairhurst, David Ros, Anna Brunstrom, Karl-Johan Grinnemo, Per Hurtig, Naeem Khademi, Michael Tüxen, Michael Welzl, Dragana Damjanovic, et al. De-ossifying the internet transport layer: A survey and future perspectives. *IEEE Communications Surveys & Tutorials*, 19(1):619–639, 2016.

[16] E. Kohler, M. Handley, and S. Floyd. Datagram congestion control protocol (dccp). RFC 4340, March 2006.

[17] Damon Wischik, Mark Handley, and Marcelo Bagnulo Braun. The resource pooling principle. *ACM SIGCOMM Computer Communication Review*, 38(5):47–52, 2008.

[18] Alan Ford, Costin Raiciu, Mark J. Handley, Olivier Bonaventure, and Christoph Paasch. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 8684, March 2020.

[19] Anwar Walid, Qiuyu Peng, Jaehyun Hwang, and Steven H. Low. Balanced Linked Adaptation Congestion Control Algorithm for MPTCP. Internet-Draft draft-walid-mptcp-congestion-control-04, Internet Engineering Task Force, January 2016. Work in Progress.

[20] Mingwei Xu, Yu Cao, and Enhuan Dong. Delay-based Congestion Control for MPTCP. Internet-Draft draft-xu-mptcp-congestion-control-05, Internet Engineering Task Force, January 2017. Work in Progress.

[21] Ramin Khalili, Nicolas Garbiel Gast, Miroslav PopoviÄ‡, and Jean-Yves Le Boudec. Opportunistic Linked-Increases Congestion Control Algorithm for MPTCP. Internet Draft draft-khalili-mptcp-congestion-control-05, IETF, Individual Submission, July 2014.

[22] Quentin De Coninck and Olivier Bonaventure. Multipath quic: Design and evaluation. In *Proceedings of the 13th international conference on emerging networking experiments and technologies*, pages 160–166, 2017.

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

[24] Per Hurtig, Karl-Johan Grinnemo, Anna Brunstrom, Simone Ferlin, Özgü Alay, and Nicolas Kuhn. Low-latency scheduling in mptcp. *IEEE/ACM Transactions on Networking*, 27(1):302–315, 2018.

[25] V Jacobson and N Kathleen. Controlling queue delay-a modern aqm is just one piece of the solution to bufferbloat. *Asscociation for Computing Machinery (ACM Queue)*, 2012.

[26] Kathleen Nichols, Van Jacobson, Andrew McGregor, and Jana Iyengar. Controlled Delay Active Queue Management. RFC 8289, January 2018.

[27] ACM. Appendix: Codel pseudocode. `https://queue.acm.org/appendices/codel.html`, last accessed on 2020-05-24.

[28] RRUL Rogues Gallery - Bufferbloat.net. `https://www.bufferbloat.net/projects/codel/wiki/RRUL_Rogues_Gallery/`, last accessed on 2020-03-09.

[29] Christian Huitema. picoquic. `https://github.com/private-octopus/picoquic`, last accessed on 2020-03-09.

[30] Tommy Pauly, Eric Kinnear, and David Schinazi. An Unreliable Datagram Extension to QUIC. Internet-Draft draft-ietf-quic-datagram-00, Internet Engineering Task Force, February 2020. Work in Progress.

[31] Mininet. `http://mininet.org/`, last accessed on 2020-04-03.

[32] Eric Dumazet. sch_fq_codel.c. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/sched/sch_fq_codel.c`, last accessed on 2020-02-22.

[33] Van Jacobson Kathleen Nichols. sch_codel.c. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/sched/sch_codel.c, last accessed on 2020-02-22.

[34] Bob Jenkins. Jenkins hash function. http://www.burtleburtle.net/bob/hash/doobs.html, last accessed on 2020-02-22.

[35] Bob Jenkins. Codel psuedocode. https://queue.acm.org/appendices/codel.html, last accessed on 2020-02-22.

[36] Injong Rhee, Lisong Xu, Sangtae Ha, Alexander Zimmermann, Lars Eggert, and Richard Scheffenegger. CUBIC for Fast Long-Distance Networks. RFC 8312, February 2018.

[37] Andrei Gurtov, Tom Henderson, Sally Floyd, and Yoshifumi Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582, April 2012.

[38] Toke Høiland-Jørgensen, Carlo Augusto Grazia, Per Hurtig, and Anna Brunstrom. Flent: The flexible network tester. In *Proceedings of the 11th EAI International Conference on Performance Evaluation Methodologies and Tools*, pages 120–125, 2017.

[39] EricssonResearch. Scream. https://github.com/EricssonResearch/scream, last accessed on 2020-05-20.

A List Of Acronyms

3GPP 3rd Generation Partnership Project

ACM Association for Computing Machinery

AQM Acitve Queue Management

ATSSS Access Traffic Steering, Switching & Splitting

BALIA Balanced Linked Adaptation Congestion Control Algorithm

BDP Bandwidth-delay Product

CDF Cumulative Distribution Function

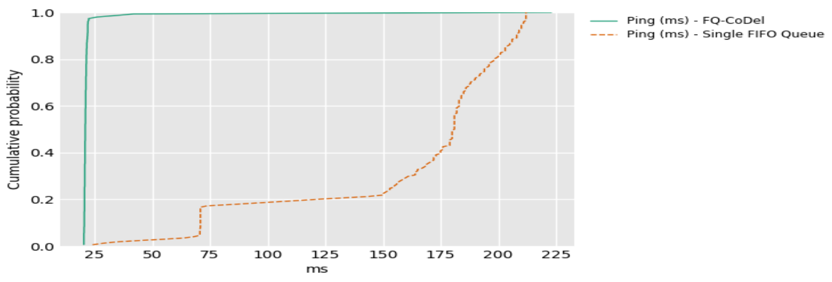
CPF	Cheapest Pipe First
CoDel	Controlled Delay
DAPS	Delay-Aware Packet Scheduler
DCCP	Datagram Congestion Control Protocol
DRR	Deficit Round-Robin
FIFO	First In First Out
FLENT	The FLExible Network Tester
FQ-CoDel	Flow Queuing Controlled Delay
ICMP	Internet Control Message Protocol
IPv4	Internet Protocol version 4
IP	Internet Protocol
LowRTT	Lowest-RTT-First
MPDCCP	Multipath Datagram Congestion Control Protocol
MPQUIC	Multipath QUIC
MPTCP	Multipath TCP
MP	Multipath
MTU	Maximum Transmission Unit
NAT	Network Adress Translation
OLIA	Opportunistic Linked-Increases Algorithm

OTIAS	Out-of-Order Transmission for In-Order Arrival Scheduler
PDU	Protocol Data Unit
PQUIC	Pluganized QUIC
PRE	Pluglet Runtime Environment
RFC	Request for Comments
RP	Retransmission and Penalization
RR	Round-Robin
SCReAM	Self-Clocked Rate Adaptation for Multimedia
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
UE	User Equipment
VPN	Virtual Private Network
wVegas	Weighted Vegas

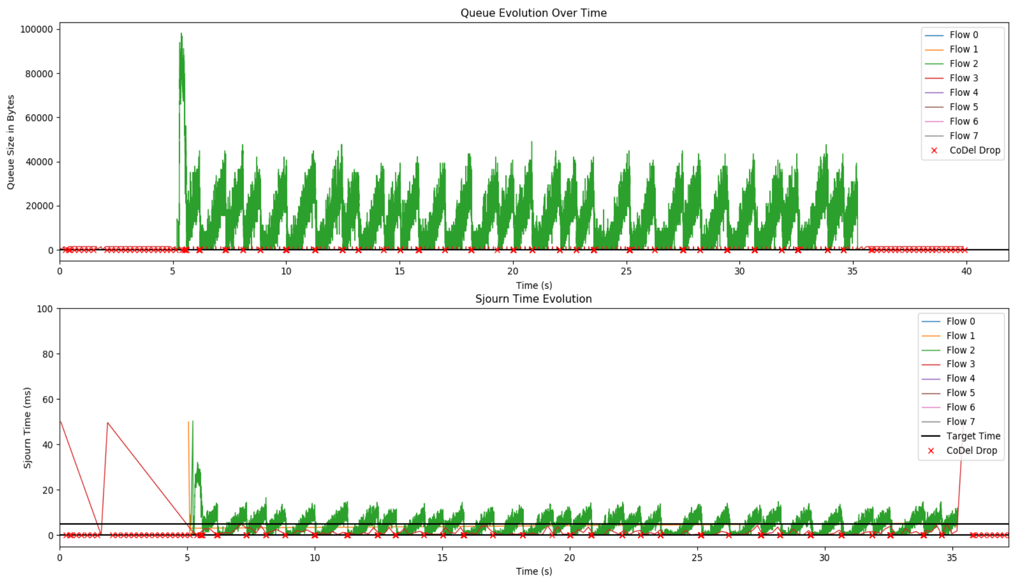
B Additional Test Results

B.1 Single TCP Flow

Figure B.1: Additional results from 1 TCP downlink flow with a concurrent downlink ICMP flow, comparing transport layer adaptation of FQ-CoDel and single FIFO queue while using the PQUIC RR packet scheduler over a homogeneous network



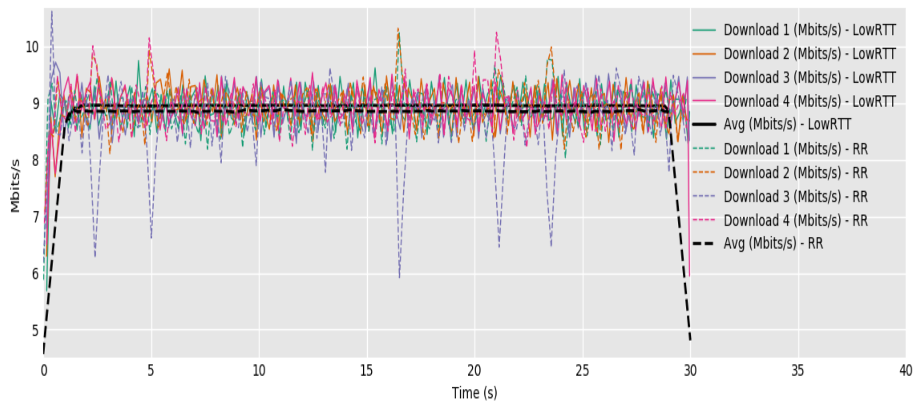
(a) Cumulative distribution of latencies comparing FQ-CoDel and Single FIFO Queue



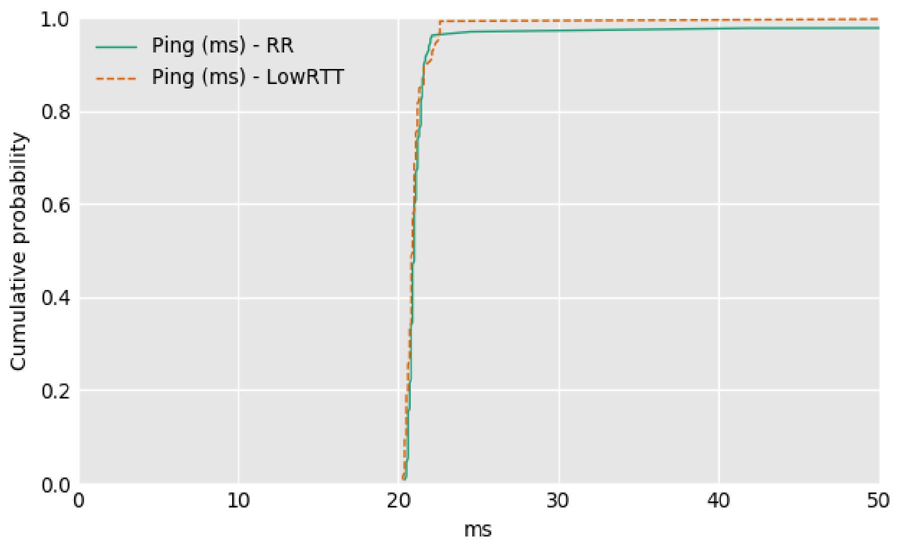
(b) FQ-CoDel queue evolution over time, the TCP flow is depicted as Flow 0, the ICMP Flow is depicted as Flow 1, the red crosses are CoDel drops and the black line indicates the target time of 5ms.

B.2 Four Concurrent TCP Flows

Figure B.2: Results from 4 TCP downlink flows and 1 concurrent ICMP flow, applying transport layer FQ-CoDel and comparing the PQUIC RR packet scheduler and PQUIC LowRTT packet scheduler over a homogeneous network



(a) Experienced throughput comparing RR and LowRTT packet scheduling



(b) Cumulative distribution of latency comparing RR and LowRTT packet scheduling

Figure B.3: Additional data including throughput results from 4 TCP downlink flows and 1 concurrent downlink ICMP flow, applying transport layer FQ-CoDel and comparing the PQUIC RR packet scheduler and PQUIC LowRTT packet scheduler over a heterogeneous network

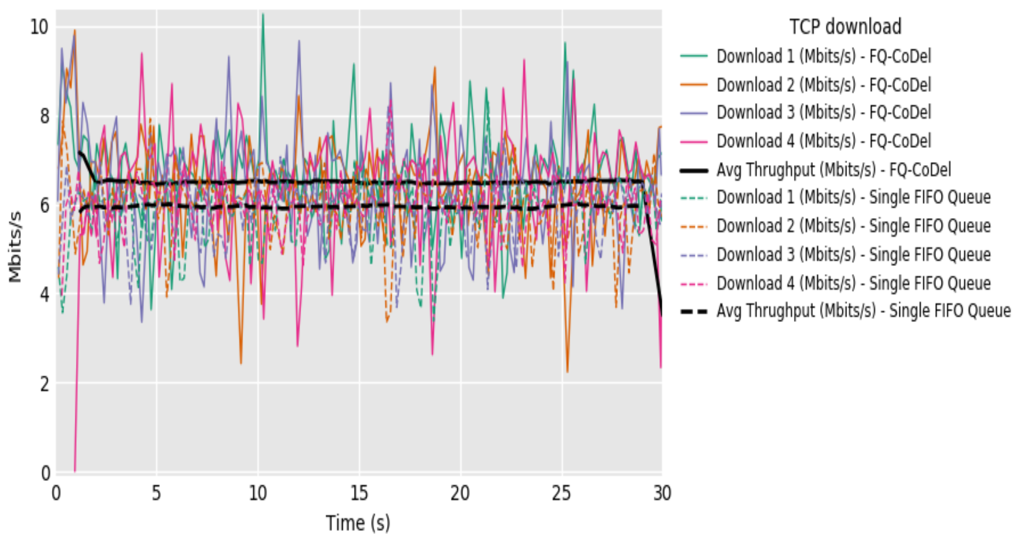
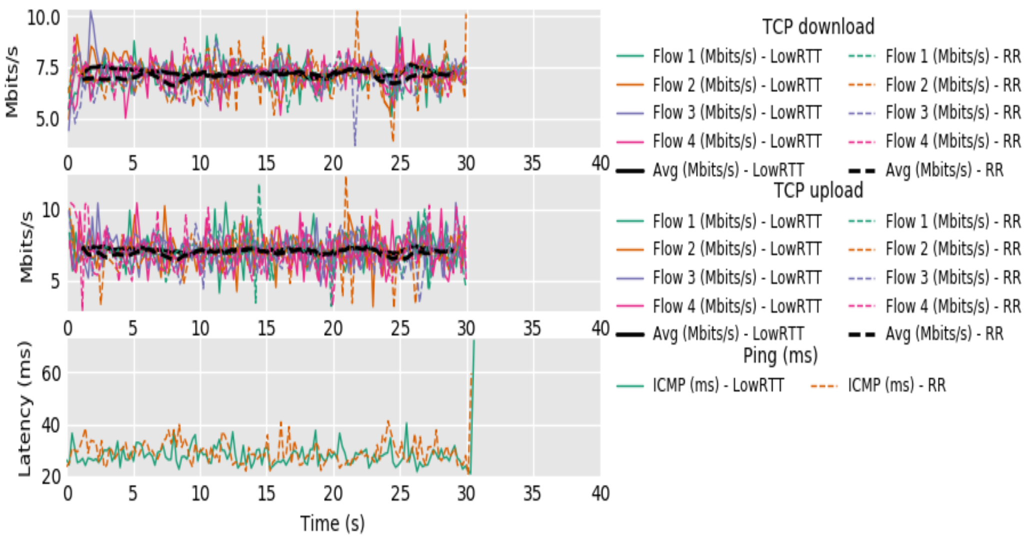
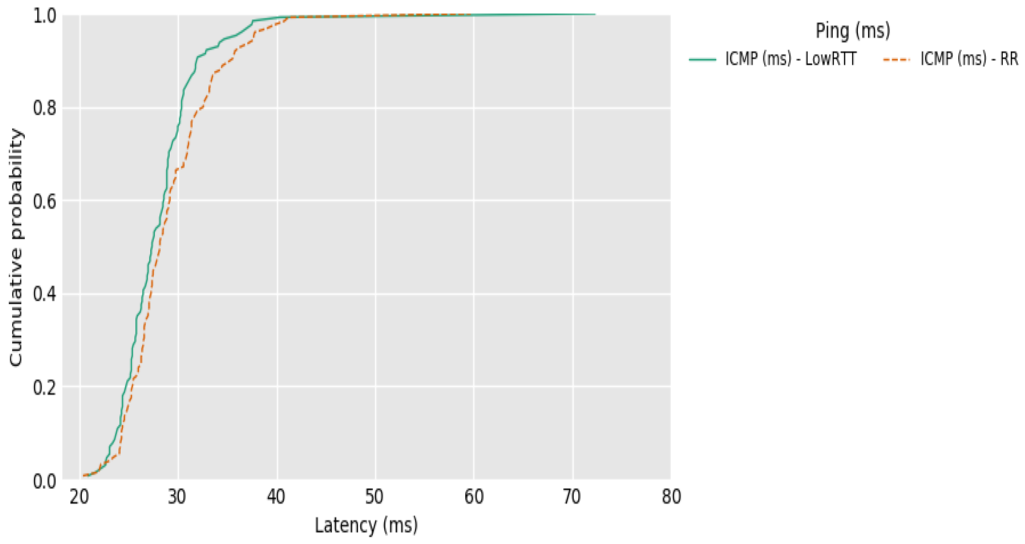


Figure B.4: Results from 4 TCP downlink flows, 4 concurrent TCP uplink flows and 1 concurrent downlink ICMP flow, applying transport layer FQ-CoDel while comparing the PQUIC RR packet scheduler and PQUIC LowRTT packet scheduler over a homogeneous network



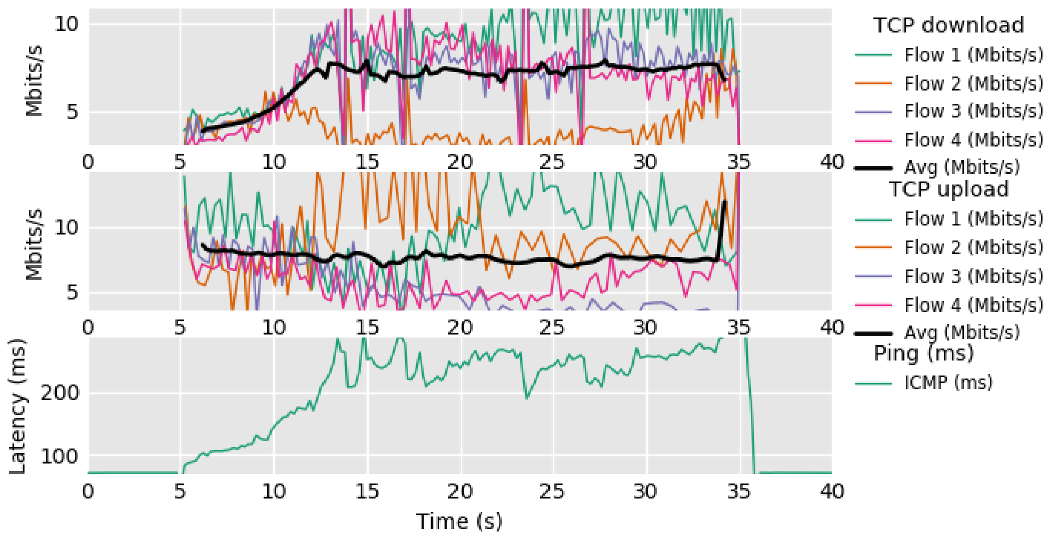
(a) Experienced throughput, both downlink and uplink, comparing RR and LowRTT packet scheduling



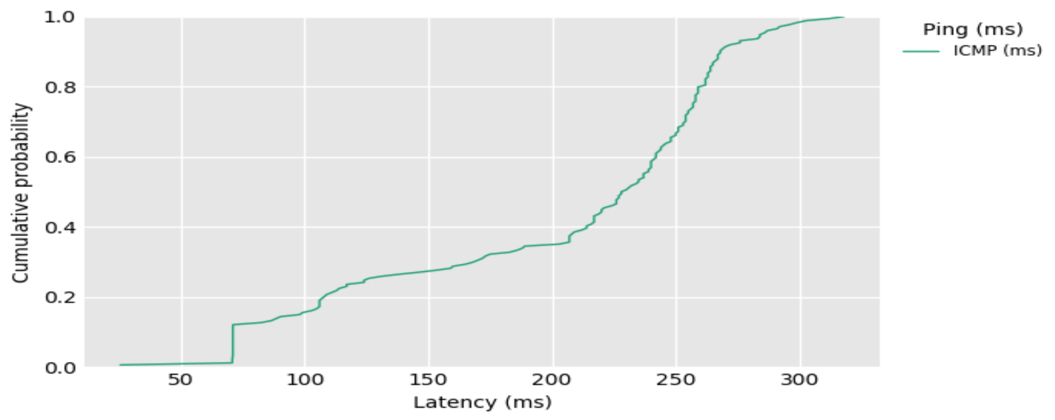
(b) Cumulative distribution of latency comparing RR and LowRTT packet scheduling

B.3 Real-time Response Under Load

Figure B.5: Results from 4 TCP downlink flows, 4 concurrent TCP uplink flows and 1 concurrent downlink ICMP flow, applying single transport layer FIFO queue while comparing the PQUIC RR packet scheduler and PQUIC LowRTT packet scheduler over a homogeneous network

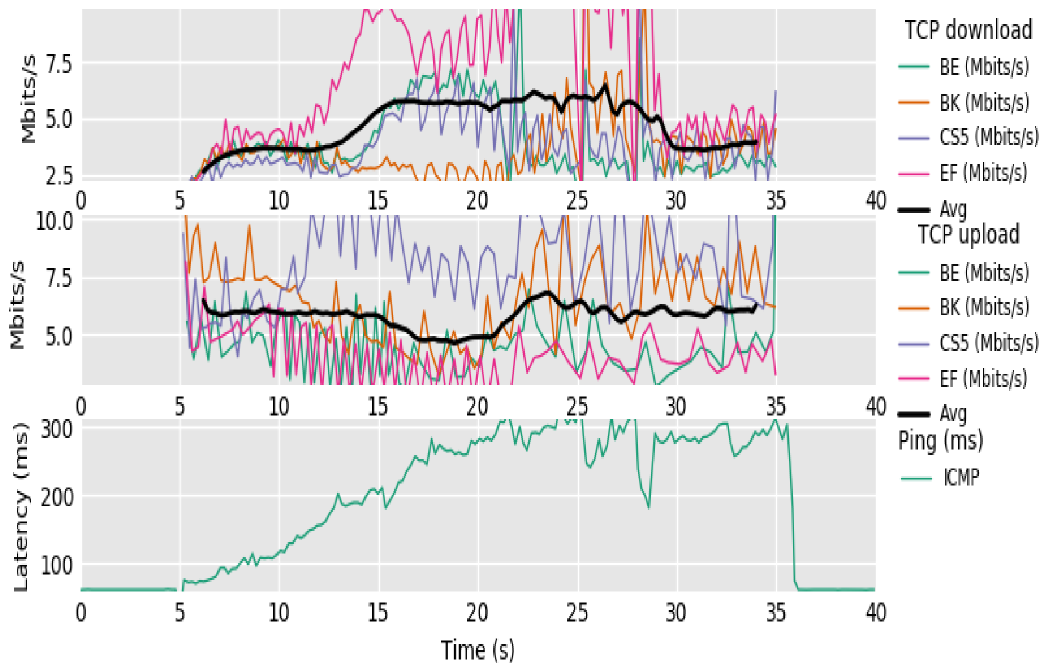


(a) Experienced throughput, both downlink and uplink, comparing RR and LowRTT packet scheduling



(b) Cumulative distribution of latency comparing RR and LowRTT packet scheduling

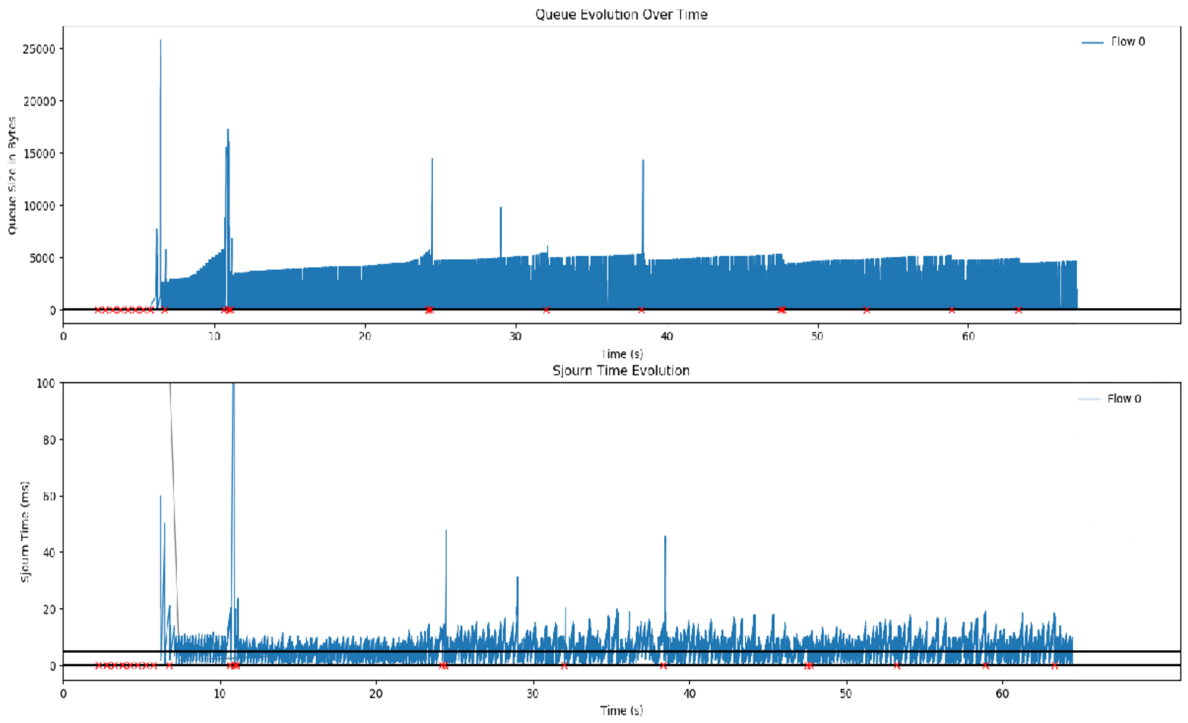
Figure B.6: Results from 4 TCP downlink flows, 4 concurrent TCP uplink flows and 1 concurrent downlink ICMP flow, applying single transport layer FIFO queue and PQUIC LowRTT packet scheduler over a heterogeneous network



(a) Experienced throughput, both downlink and uplink, comparing RR and LowRTT packet scheduling

B.4 PQUIC VPN Problem

Figure B.7: Addition results from 1 SCReAM downlink flow, applying transport layer FQ-CoDel and the PQUIC RR packet over a homogeneous network



C Datagram Plugin Manifest

Algorithm 4 Datagram Plugin Manifest

```
be.mpiraux.datagram
parse_frame param 0x2c replaceparse_datagram_frame.o
parse_frame param 0x2d replaceparse_datagram_frame.o
parse_frame param 0x2e replaceparse_datagram_frame.o
parse_frame param 0x2f replaceparse_datagram_frame.o
write_frame param 0x2c replacewrite_datagram_frame.o
write_frame param 0x2d replacewrite_datagram_frame.o
write_frame param 0x2e replacewrite_datagram_frame.o
write_frame param 0x2f replacewrite_datagram_frame.o
write_frame param 0x60 replacewrite_dummy_frame.o
process_frame param 0x2c replaceprocess_datagram_frame.o
process_frame param 0x2d replaceprocess_datagram_frame.o
process_frame param 0x2e replaceprocess_datagram_frame.o
process_frame param 0x2f replaceprocess_datagram_frame.o
notify_frame param 0x2c replacenotify_datagram_frame.o
notify_frame param 0x2d replacenotify_datagram_frame.o
notify_frame param 0x2e replacenotify_datagram_frame.o
notify_frame param 0x2f replacenotify_datagram_frame.o
notify_frame param 0x60 replacenotify_datagram_frame.o
connection_state_changed post cnx_state_changed.o
send_message extern send_datagram.o
ge_message_socket extern get_datagram_socket.o
get_max_message_size extern get_max_datagram_size.o
prepare_packet_ready pre process_datagram_buffer.o
```