UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

New applications above QUIC

Dissertation presented by
Clarembeau Alexis, Floriot Rémi

for obtaining the Master's degree in
Computer Engineering and Computer Science

Supervisor(s)
Bonaventure Olivier

Reader(s)
De Coninck Quentin, Sadre Ramin

Academic year 2017-2018

Thanks

Many people have contributed either directly or indirectly to this study.
In particular, we would like to thank professor Bonaventure and Quentin De Coninck for their helpful support all over the writing of this document. Their clear vision and practical knowledge has been a valuable asset to work with QUIC.
We also wanted to thank all the INGI department, our friends, François and Maxime and our family for their contribution, at various levels.

Contents

Abstract

QUIC is an emerging network protocol announced publicly in 2013 by Google. Initially developed to improve HTTP/2, QUIC provides faster reliable transmissions by defining a new transport layer, which includes better loss handling, stream multiplexing and built-in secure communications over a single transport layer over UDP.

This work explore how QUIC can benefit to other applications. It presents QuicSSH and QuicVPN, as QUIC based clones of two common tools: SSH and VPN, which generally run over TCP or UDP. As SSH, QuicSSH is a software offering remote login and port forwarding. It can execute commands or forward connections to a distant network using a secure protocol. QuicVPN creates virtual private networks. By operating at the IP layer, QuicVPN encapsulates IP datagrams into QUIC packets. Then, it relays them from one network to another to make them appear as "connected".

We show that both QuicSSH and QuicVPN improve their TCP or UDP equivalents by exploiting the features of QUIC such as: its ability to multiplex streams and its improved retransmission mechanism. Using a in-depth experimentation in a wide range of possible networks, we identify the strengths and weaknesses of both programs. Despite we highlight some problems in our QUIC implementation, we prove that QuicSSH and QuicVPN outperform their TCP-based alternatives in many aspects.

Finally, thanks the user space nature of QUIC, which make it easily extensible, we also add new features. We see how multipath capabilities can benefit to QuicSSH and allow it to perform bandwidth aggregation and faster handover. By trying different scheduling policies, we found some flaws in the multipath QUIC implementation, which make it sometimes less efficient than a single path implementation. We propose a new system to enhance the reactivity of reliable tunnels in QuicVPN. By using a mechanism based on explicit congestion notification, we prove how it can avoid the collapse of such VPN in variable bandwidth networks and generally achieve lower latency. We also present an extension to make dynamic QoS by prioritizing QUIC streams at the application level.

Chapter 1

Introduction

HTTP [1] is the most widespread protocol over the internet, accounting for more than 60% of the residential communications [2]. Designed in the early 1990s, this protocol was improved several times to meet the requirements of today's internet. But, increasing needs for better performance, especially for growing mobile networks, cannot just be addressed in the application layer. TCP [3], which is the underlying transport protocol of HTTP, also has problems. The impossibility to migrate from one network to another, the excessive latency when establishing a secure connection over TLS, the lack of built-in data multiplexing as well as the poor quality of the ACK mechanism have an impact on the performances of HTTP, even on its newest versions [4]. Moreover, all of these elements are combined with the extreme difficulty to improve TCP itself. Due to its implementation in the operating system kernel, it is very hard to modify. And, any change in the protocol can interfere with the middle-boxes, which sometimes defines strong expectation on the TCP behaviour.

Knowing this, Google designed a new transport protocol: QUIC [5]. To solve all of these problems, QUIC merges more efficient transport, cryptography and reliable data transmission in an easily extensible single layer user space implementation on top of UDP. Already deployed in their Chrome browser for some parts of the web traffic, QUIC is gaining the interest of many other companies and of the IETF, which is starting to standardise the protocol.

But we think that QUIC protocol can go beyond HTTP and replace TCP in many use cases. As core components of many enterprise networks, we explore the possibility to use QUIC as a transport layer for two other applications: SSH and VPNs. By building functional clones named QuicSSH and QuicVPN, we will assess whether QUIC can be a good fit for these tools. Using a strong experimental framework, we compare the performance of these applications built on top of QUIC to their TCP equivalents. We go even further and try to improve those by adding new features aiming to increase their security, efficiency, reactivity or multiple path capabilities.

In this paper, we first describe the background of the protocols. We answer the following questions, that are: What is QUIC?, What is SSH? And what is a VPN? We also state the best practises for rigorous performance measurements. Then, in Chapter 3, we introduce the methodology we follow to assess the efficiency of our programs: infrastructure, network characteristics and experimental design procedure. Chapter 4 describes a required feature missing in the QUIC implementation we use: client authentication. The next two chapters (5 and 6) present our new applications, their architecture and performance. We conclude this document by a critical review of what we have learned when building our applications. By looking at their benefits and defaults, we see how and where our programs can replace their original TCP equivalents. We also give some tracks to improve QuicSSH, QuicVPN, the QUIC protocol itself and its extensions.

Chapter 2

State of the art

Before introducing the principles of our applications, we start by reviewing the current state of the literature about QUIC, SSH and VPNs as well as the best practices for experimenting with network applications.

2.1 QUIC

QUIC is a new transport protocol initially designed by Jim Roskind at Google and presented for the first time in 2013 [5]. Its goal is to provide fast and secure communications between hosts using a user space implementation over UDP. Initially developed to improve HTTP/2 for Google products, it is now becoming a generic purpose transport protocol under an ongoing standardisation process at the IETF [6].

There are two major variants of QUIC: one from Google, called gQUIC and one upcoming, for general purpose, that is called IETF QUIC. Over time, gQUIC should implement the specification of IETF quic, but there are still some differences. In this paper, we will refer as QUIC for the gQUIC implementation that we use [7].

2.1.1 Principles

The basic principles of QUIC are similar to any other reliable transport, such as TCP. Both protocols aim to establish a reliable data transport channel between two hosts. Both use re-transmittable packets with congestion control and a connection establishment. However, QUIC provides various improvements that make it different from TCP.

Connection establishment

QUIC's connection establishment tries to reduce as much as possible the number of round trip times needed for initial handshake. By integrating the cryptographic setup as soon as possible in the connection agreement, QUIC trims down the number of round trip times needed from 3 in a TCP+TLS session up to a direct 0 RTT transmission .

In the IETF specification, this is implemented using a variation of TLS 1.3 [8]. But, historically gQUIC used a custom protocol [9]. Here, we only discuss the gQUIC version (Figure 2.1).

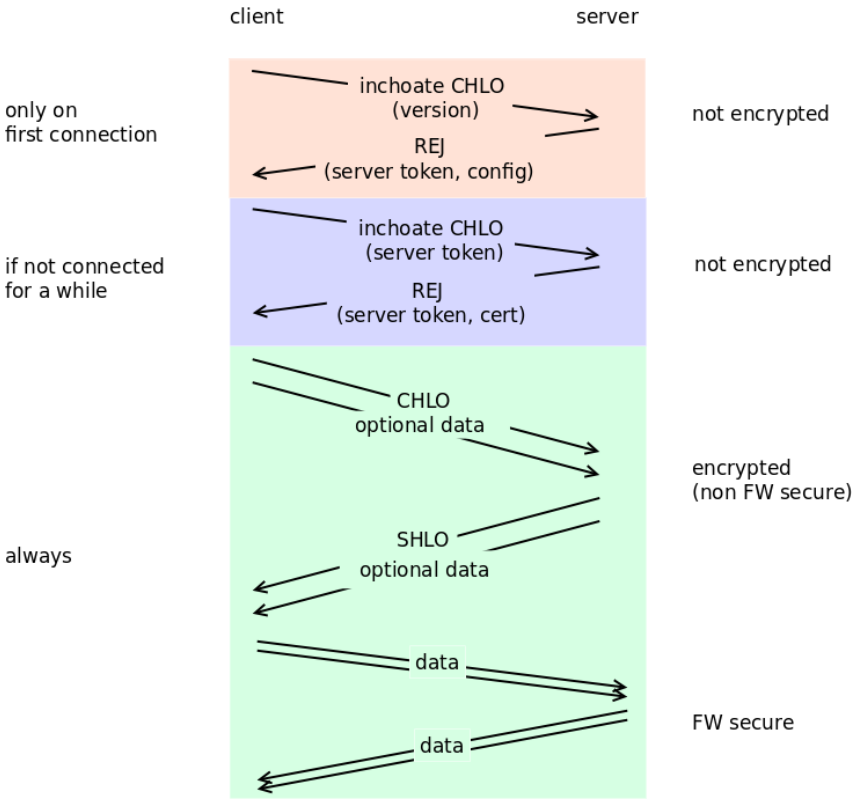

Figure 2.1: Illustration of a 2-RTT (g)QUIC handshake

In the worst case, gQUIC first connects to a server it has never seen. It starts by requesting the server configuration and a token thanks to a round of CHLO-REJ messages. Then, using another round, it request the server certificate chain. After two RTTs, the client is thus able to communicate to the server with derived long-term keys. But, this is not forward-secure. This means that if this key is compromised, all the future and past communications can be decrypted. That is why the protocol adds the negotiation of an ephemeral secret in the CHLO and SHLO message. Depending on the client needs in terms of security, it can directly send message at this moment or wait to have finished the CHLO-SHLO exchange [10].

Most of the improvements of QUIC come when the client was previously connected to the server. In that case, QUIC clients remembers the server configuration. It skips the first (red) phase and directly starts by serving its token. If the previous connection was not very recent, the token is rejected. The client then gets a new one and can start to communicate in a non-forward secure way after 1 RTT.

But there is also a third case. When a gQUIC client connects to a server just after a previous connection, it can use a mechanism called "session resumption", or also "0-RTT". In that case, the server token is still valid. QUIC can skip the first two steps and directly start communicating in a non-forward-secure way.

Data transmission

QUIC also provides fast and reliable data transmission. It solves many well-known problems of TCP that we cannot easily change due to the wide adoption of the protocol. It implements monotonously increasing sequence numbers to avoid TCP ack ambiguity [11] . It adds more ACK ranges compared to TCP, it introduces an explicit value of the delay and other mechanisms for better retransmission of packets in many cases, such as in the case of tail losses [12].

Moreover, by its user space implementation, QUIC allows faster testing and deployment of various mechanisms, which can help to achieve better efficiency. A good example is on the

congestion control. Currently, nearly all QUIC implementations, which base on Google code use Cubic [13] as default[1], which is a common loss-based congestion control used in the TCP stack of the Linux kernel. But some of them even experiment BBR [14]. This new algorithm uses an explicit model of the network queues and congestion state based on measurements of bandwidth and delay. Instead of basing on losses, BBR grows progressively the congestion window until the round-trip delay starts to increase. This better handles spurious losses, as they do not impact the congestion window. Moreover, this also helps to avoids a phenomenon known as "bufferbloat" [15], which impacts negatively interactive applications. As shown on Figure 2.2, loss-based congestion controls stabilise when they completely fill the buffers of routers (B). That is not the case with BBR. This protocol stops sending faster when growing the congestion window increases the RTT without changing the delivery rate. This allows to reach a much more optimal operating point (A), which has the same bandwidth, but a much lower delay.

If BBR is confirmed to be faster than Cubic, programs using QUIC can quickly deploy it simply by updating their QUIC library. In TCP, this would require a support from all operating systems, which takes years.

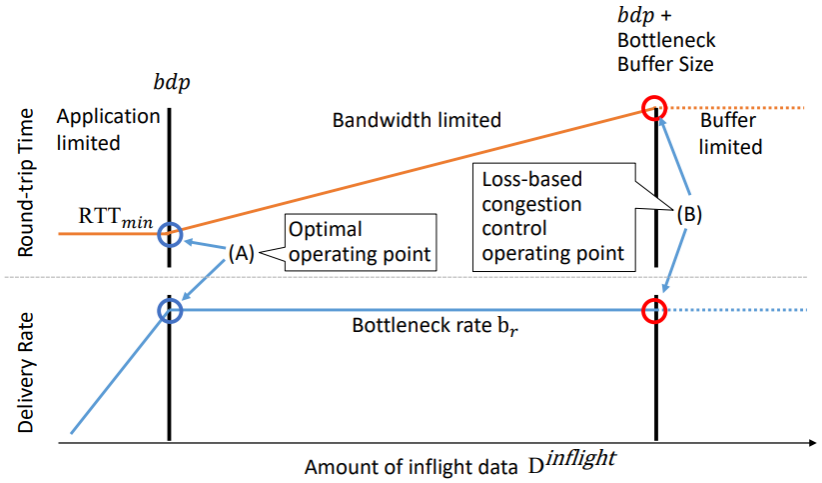

Figure 2.2: Illustration of BBR optimal operating point (A) compared to a loss-based congestion control (B) (such as Cubic) [16] © 2017 IEEE

Connection migration

QUIC also allows hosts to migrate between different networks and change their IP addresses. Thanks to the unique connection identifier in the public header, QUIC sessions are independent of the classical TCP endpoint identifiers (IP & port source & destination) making network handovers possible. This is particularly helpful if a QUIC client is moving from one network to another, or if it is subject to NAT rebinding.

Multiplexing

QUIC has a built-in multiplexing mechanism which allows bundling multiple data flows inside a single QUIC session by defining a notion of streams. This helps, when data is lost on one stream, to allow all other streams to run without waiting for retransmission, avoiding a phenomenon known as "head-of-line blocking".

[1] `https://cs.chromium.org/chromium/src/net/third_party/quic/core/quic_flags_list.h?q=quic_default_to_bbr&dr=C`

As illustrated on Figure 2.3, when carrying multiple flows inside a single TCP session, any packet lost on any flow blocks the entire session. With QUIC, each flow gets its own independent stream, which does not block the others.

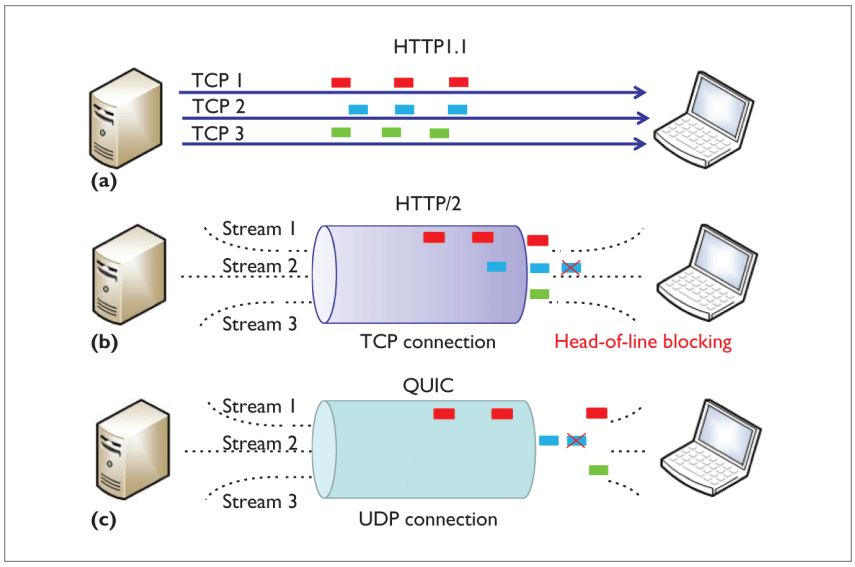

Figure 2.3: Illustration of head-of-line blocking [17] © 2017 IEEE

Termination

As in TCP, QUIC provides two ways to terminate a connection. But it adds flexibility by allowing this at the session or at the stream level. QUIC sessions finish either when they are idle for more than 30 seconds or when a `CONNECTION_CLOSE` frame is sent. In both cases, connection is abruptly closed and some data may not be re-transmitted. Streams can be closed in two ways: either we can gracefully stop sending in one direction using FIN bit or we could also use RST to abruptly finish the stream. Of course, when we terminate a session, all QUIC streams are automatically closed.

2.1.2 Packet format

To support those features, QUIC defines its own packet format using a flexible internal structure defined over UDP datagrams (Figure 2.4). This choice comes from many reasons. This avoids the need to deploy a completely new protocol, which would be hard to deploy. Moreover, UDP is a well-known protocol, which already supports numerous other applications and easily passes through firewalls.

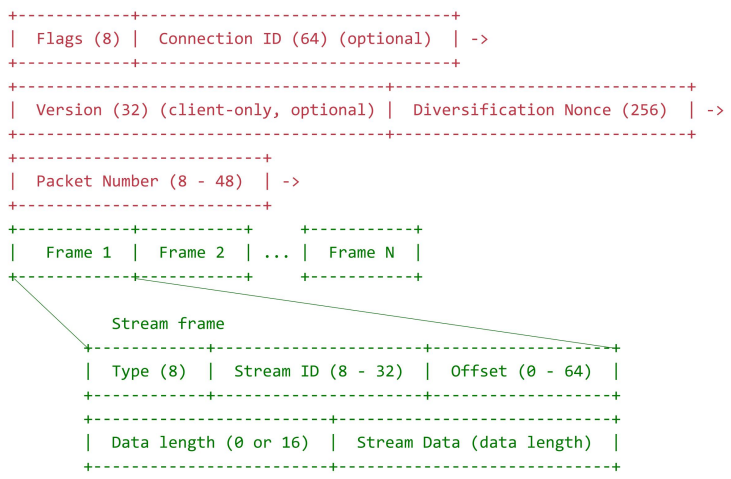

Figure 2.4: Illustration of a QUIC frame packet containing a stream frame [18]

The first part of the QUIC payload defines public, unencrypted informations such as: a sequence number, a unique connection identifier and a generic flag [19]. The flag indicates which type of data packets we are using: mainly version packet or frame packet. Version packet helps to set up an agreement on the version of QUIC to use. Frame packets carry the useful data of QUIC.

In the case of the frame packet, the last part of the UDP payload contains an authenticated and encrypted private part. It secures a list of "QUIC frames". They can contain application data (stream frame), acknowledgements (ack frame) or congestion control information (stop waiting and congestion window update frame) and many others (ping frame, padding, connection close, stream blocked, ...).

Stream frames give information about the identifier of the stream carrying the data, the offset inside this stream, the payload length and its content. Stream creation does not require any new frame. A new stream is implicitly opened when receiving a new stream frame containing an unknown stream ID. All the others are used for the protocol itself. They can be used to close a connection, a stream, check the connection state, tune the congestion window, indicate losses. Implementers can even decide to create their own frames to support specific transport needs.

2.1.3 Implementation

Despite its relatively young age, QUIC has already been implemented in different client and server applications. Google was first to provide a home-made version of the protocol, before the publication of its first specifications in 2013. They built a C++ implementation used in their Chrome web browser and in production servers to power services such as Gmail or YouTube. Using the open source code of the Chromium project[2], several developers started to extract QUIC protocol code to turn it into an independent library[3].

In this document, we use a native go implementation that is compatible with Chromium code called `quic-go`[4]. This choice comes from the great quality, readability, simplicity and extensibility of the code, which allows easy customisations.

Even though the code is not optimized for performance and does not implement the full protocol features, such as 0RTT handshake or path MTU discovery, its large contributor base and inspiration from Chromium code led us consider it as suitable for research purpose.

[2] `http://www.chromium.org/developers/how-tos/get-the-code`

[3] `https://github.com/devsisters/libquic`

[4] `https://github.com/lucas-clemente/quic-go`

version 0.7, for the entire thesis, excepted the chapter on QuicSSH (version 0.6)

2.1.4 Interface

In terms of interface, `quic-go` provides the common functions that all other transport protocol propose. It allows to open and listen for connections as well as to close them. It can open, accept or close streams, read and write data on them.

The only elements which differ compared to common TCP interfaces is the lack of buffers. Due to its internal design, each write operation on a QUIC stream does not complete immediately, but when the data is sent over the network. This is used later, when we implement our queuing systems. It is also important to take it into account when building programs for performance. When in TCP, two subsequent small writes will most often result in the emission of a single packet, this is not the case for our QUIC implementation. `quic-go` does not support the naggle algorithm. So, each write directly reflects to the network.

2.2 SSH

Secure Shell (SSH) is a protocol that provides secure communications over a possibly insecure network [20]. With this protocol, confidentiality, authentication and integrity is ensured without making any assumption on the underlying network. Several services are available in SSH. We mainly explore the remote login and the port forwarding.

SSH protocol is composed of three parts (Illustrated in Figure 2.5). The *Transport Layer Protocol* ensures integrity and confidentiality. The *User Authentication Protocol* lets the server authenticate the client (via a password or a key authentication) before accepting any service request. Finally, the *Connection Protocol* provides services, like remote login and port forwarding, that can run simultaneously using logical channels multiplexed over a single connection.

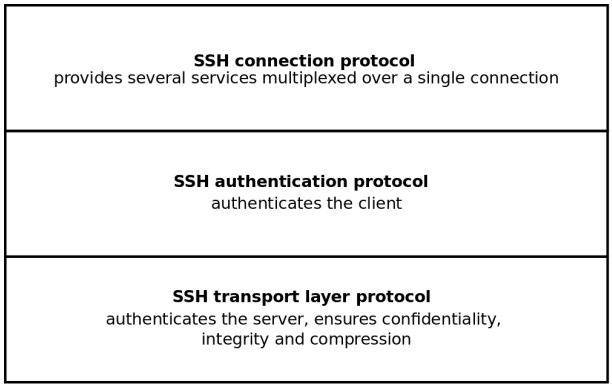

Figure 2.5: Illustration of the three SSH layers

In SSH, each server should have at least one host key. The server can provide its public key to the client to prove its identity. There are two possible ways for the client to verify this public key: either the client stores locally the public keys of trusted servers or it uses a mechanism of certificate chain verification. Currently there is no well-established key infrastructure for SSH, in contrary to what we can find for example with TLS. So, SSH clients generally use the first option. The main drawback when using a simple local database to store the public keys of servers is that the first connection to a server is not secure enough against active man-in-the-middle attack. Anyone could intercept connection requests and answer with its own host key.

2.2.1 SSH Transport Layer Protocol

This first layer in the SSH Protocol establishes a session between the client and the server to ensure confidentiality and integrity for the exchanged messages. It also provides server authentication [21].

Any underlying transport protocol that protects against transmission errors is suitable to establish SSH connections. The most common choice is TCP on port number 22, which is officially registered for this purpose by the Internet Assigned Numbers Authority (IANA).

When a connection is initiated between a client and a server with SSH, we need to negotiate several elements: the key exchange algorithm, the encryption algorithm (aes128-cbc [5] is recommended in RFC4253 [21]) and also the MAC algorithm for data integrity (hmac-sha1-96[6] is recommended in RFC4253 [21]).

By using Diffie-Hellman key establishment protocol, we share a common secret to derive encryption key and MAC key. This algorithm ensures perfect forward secrecy [22]. It means that if long term keys (host keys in SSH) are compromised, this does not impact the previous communications using session keys. Key establishment is followed by server authentication by using a signature with host key.

2.2.2 SSH Authentication Protocol

This second layer comes on top of the previous one. It assumes the previous establishment of a connection ensuring confidentiality, integrity and server authentication. This layer provides client authentication [23]. Several methods to authenticate the client to the server are available. We describe here the two most used: public key authentication and password authentication.

In public key authentication, the public key of the client is included in a list of allowed keys on the server. The possession of the corresponding private key demonstrates the client identity[7]: a signature, with the private key, of some fields specific to the current connection (session id, user name, service name, ...) is sent to the server which can verify it using the public key it already knows.

For the second authentication method available in SSH, the client sends to the server a username and a password entered by the user. The already established Transport Layer ensures the confidentiality of those sensitive information. On the server side, we check if the given username is allowed to connect and if the given password matches with the one stored for this username (generally using secure hashes to avoid storing clear-text passwords).

2.2.3 SSH Connection Protocol

This third and last layer, built on top of the two previous ones, provides several services such as remote login and port forwarding. Those can run simultaneously and bundle up in logical channels multiplexed over a single encrypted connection [24].

The remote login is a tool that allows a user to remotely execute commands in a terminal window and see the corresponding output, as if the user was working on the remote machine directly.

Port forwarding is a technique that interrupts a connection between a client and a server. It ends a connection on a first intermediate host (the SSH client) and then reopens another corresponding connection on a second intermediate host (the SSH server). The packets captured on the first intermediate host are transferred towards the second intermediate host using the SSH connection. Then, they are forwarded to the destination. The same mechanism applies to the reverse path for the server responses.

Figure 2.6 shows an example. In this example, host A wants to download an HTML page served by a host named S. But there is a firewall that prevents this connection (for instance S is a web server of a company that is accessible only for its employees). However, another host B is able to query the server S and A can connect through SSH to B. A can thus send to B a

[5] AES symmetric bloc cipher in cipher block chaining mode, with a 128-bits secret key

[6] Hash based Message Authentication Code using SHA-1, truncated to 96 bits

[7] Actually, often, owning the private key is not enough to be authenticated as a password can be set-up to be able to sign something with the key.

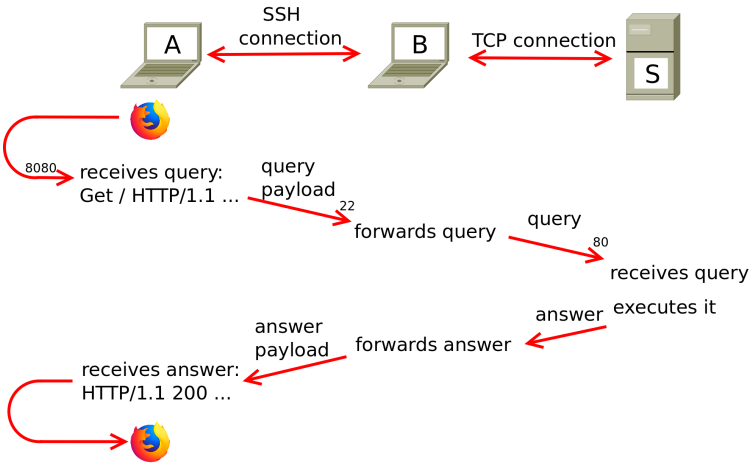

Figure 2.6: Port forwarding schema

port forwarding request to transfer all packets ending on A on port 8080 to the server S on port 80. When this is in place, opening a browser at the location `localhost:8080` on A shows the desired page, because host B initiated the connection.

There are two types of port forwarding: local port forwarding and remote port forwarding. In local port forwarding illustrated above, the client asks to the server to forward connections ending on the client. With remote port forwarding the client asks to the server to forward connections ending on the server host to the client. This can, for instance, make a local resource accessible, by listening and forwarding connections from a public host.

2.3 VPN

A VPN is tool that allows building private connections between distant hosts separated by a public or shared medium.

Initially developed to connect remote branches to main sites in an enterprise network, the VPNs can also have many other applications: improve communication security, bypass geo-restrictions, access resources in a private network, ...

They work by establishing point-to-point connections between remote VPN endpoints, forming what we call a "tunnel" (as illustrated on Figure 2.7). To connect distant networks, endpoints read data coming in each internal network and send it over this virtual link. Technically, the "tunnel" can use any transmission technique that allows forwarding data from one host to another. Depending on the technology used for the transmission and the layer of data to transfer, there are thus many kinds of VPNs. The most popular are layer 2 and layer 3 VPNs, which relay Ethernet or IP datagrams. And, they can use any transport mechanism, such as: TCP, UDP, IPsec, MPLS, ...

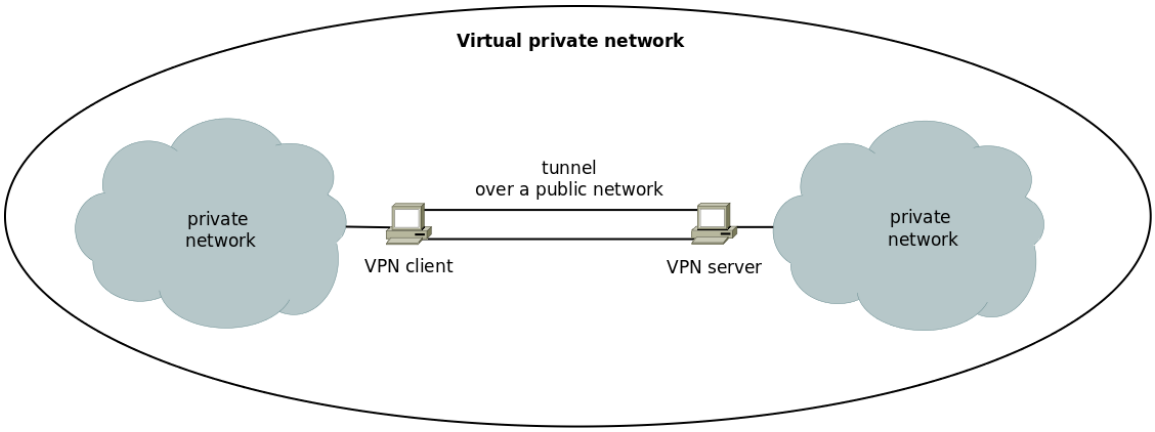

Figure 2.7: Illustration of a VPN

2.3.1 TCP/UDP based VPNs

Among the broad range of existing VPNs, this document focus on one particular kind: layer 3 TCP or UDP based VPNs. Those use standard TCP or UDP connection to establish a point-to-point link that relays IP datagrams.

Both establish a VPN by opening a session to a distant server. If needed, they authenticate each endpoint and establish a transmission channel, which can be secured, either using DTLS[25] for UDP or TLS for TCP, sometimes combined with custom security protocols. Then, they read data incoming from the network and push it into TCP or UDP sessions.

2.3.2 TCP Meltdown

Despite all the VPNs expose similar features, their interaction with the transported traffic is different. Cryptography can reduce the throughput on CPU-limited hosts. Transmission layer can truncate packets, leading to fragmentation. And, adding reliability in the transport layer can have an impact on global performance.

By handling losses and congestion control at the overlay level, reliable tunnels, that for instance use TCP, expose to the applications inside the tunnel a "perfect virtual link" that never sees reordering or losses. This can be an advantage if the tunnel is running a good TCP implementation, which is able to retransmit very fast, or if the traffic inside the tunnel does not cope very well with losses. They can also play a role in large network infrastructures [26] to control the fairness of a set of bundled connections.

But, they can also cause some problems. TCP induces a complexity overhead in the protocol. It has bigger headers, which reduce the remaining size for the useful payload. It can also interact badly with the transported traffic. This interaction, generally observed when running TCP sessions inside a TCP tunnel is commonly known as TCP collapse, or "meltdown" [27].

Analysis

To identify the kind of problems that can occur, let's consider two hosts establishing a TCP session that directly goes inside a TCP tunnel (as represented in Figure 2.8). In that case, we identify two consequences of using a reliable TCP tunnel.

a) Inadequate bandwidth

First, it could lead to a perturbation of the transported TCP sessions congestion window. In most of TCP implementations, hosts lower their sending rate each time they detect a loss. But, as for transported TCP sessions, the "tunnel" link looks perfect, this will never happen. Hosts

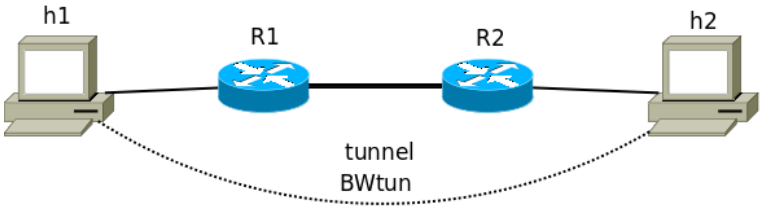

Figure 2.8: Illustration of a VPN running in our test setup

will start to send faster than the VPN endpoints, which can only exchange data at BW_{tun}. So, they will have to create a queue. This queue induces increasing delays that will only drop when the TCP retransmission timers expire.

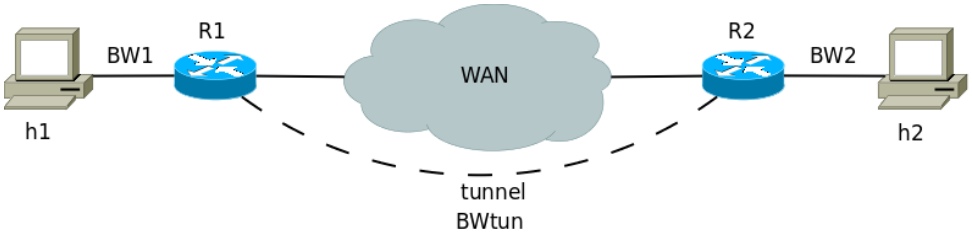

Figure 2.9: Sample network, where the VPN is not initiated by the end hosts

And, even in more complex networks, where the traffic does not directly go through the VPN (Figure 2.9), we have the same problem. TCP adapt its congestion window to losses appearing on links outside the tunnel (from hosts to routers). This makes TCP able to tune its bandwidth to

$$BW_{send} = \min(BW_1, BW_2)$$

If the bottleneck is on the access link ($\min(BW_1, BW_2) < BW_{tun}$), the VPN does not create any queue. But if the bottleneck bandwidth is on the tunnel, the VPN will accept packets at a rate that is $\min(BW_1, BW_2)$. As it can only send packets at BW_{tun}, it will also have to create one queue. This still leads to growing delay until timer expirations, which harms the reactivity of some application.

To solve this problem, network engineers often assumed that TCP based VPNs can mostly work over networks with excess bandwidth on the WAN network [28] ($BW_{tun} > BW_1$ and $BW_{tun} > BW_2$). But this greatly reduces the deployment of reliable VPNs. Some others decided to shape networks to have excess bandwidth on the WAN by limiting the host to router link ($BW_1 = BW_2 = BW_{tun}$). But, even if that works for networks with fixed bandwidth, WAN links are shared and dynamic. Bandwidth often vary over time. If we are not able to predict the bandwidth, shaping must be applied at a level lower than the minimal WAN speed and set $BW_1 = BW_2 = min_t(BW_{tun}(t))$, wasting a lot of unexploited bandwidth. There are also others solutions, such as to require all clients to use non-loss-based congestion controls as Vegas [29] or BBR [30] inside the tunnel. Despite it would effectively reduce the meltdown, this requires a full control of all machines within a network, which is often unrealistic.

TCP based VPNs are thus either deployed with care, only on specifically tuned networks, or used while accepting the possibility of an eventual meltdown.

b) Unnecessary retransmissions

There is also a second problem if the tunnel link is lossy at a point that the VPN TCP retransmission timer (RTO) often expires. Since the RTO doubles each times it expires [31], it can happen that the transported sessions have timers much smaller than the TCP session of the

tunnel. In that case, the inner TCP session can start to retransmit packet even before the VPN TCP session already remarked the loss. Depending on the difference of the value of the timers, many packets can be unnecessary retransmitted and thus block all the transported sessions.

Hopefully, this problem, despite looking technically complex to solve, does not really harm transported traffic on modern networks. Some studies [32] shown that, when both the transported traffic and the tunnelling technology support selective acknowledgement, TCP is often able to recover losses quickly enough to avoid such retransmissions to occur.

2.4 Performance measurement

Performance measurements is also a core topic when it comes to comparing the efficiency of different network applications.

Earlier experiments with MPTCP gave us some insight about how to properly characterise a transport protocol [33]. We cannot restrict performance measurements to a specific set-up with associated links having a fixed bandwidth, loss rate or delay. We need to consider the diversity of network conditions existing on the internet in order to provide valuable results. Such proper comparison requires also to perform a combination of active simulations, experimentations, traffic generations and live traffic measurements in order to measure values close to real performance.

As we cannot consider all combinations of link parameters and topologies, we use in this document a technique called experimental design. It is useful to reduce search space while keeping accurate results [34]. Also for each iteration of any experiment, we cannot be satisfied with one single measurement. Indeed this could not be statistically relevant enough. We thus prefer performing measurements several times in order to increase results significance.

Chapter 3

Methodology

3.1 Infrastructure

To be able correctly compare our applications, we need to set up an infrastructure allowing repeatable and reliable results. In this section we discuss the hardware, framework and tools we put in place to achieve this.

3.1.1 Hardware

The workstation we use for the experiments has the following specifications. Its CPU is a Intel Core 2 Duo E8600 (64 bits, 3.33Ghz). It has 4GB of DDR2 RAM and comes with a local 1TB hard drive disk. It runs under the Fedora 27 Linux distribution with the version 4.13.13 of the kernel.

We keep all the default network settings, especially for TCP, which includes the "cubic" congestion control, pacing, selective acknowledgement, early retransmissions, socket buffers tuned based on the amount of RAM, ... It is important to notice that despite the fact that this Linux version supports a TCP BBR implementation, we don't use it. As this document mostly focus on the applicability of QUIC, we keep the default parameters for both protocols to use Cubic as the congestion control, both for `quic-go` and the TCP stack.

3.1.2 Framework

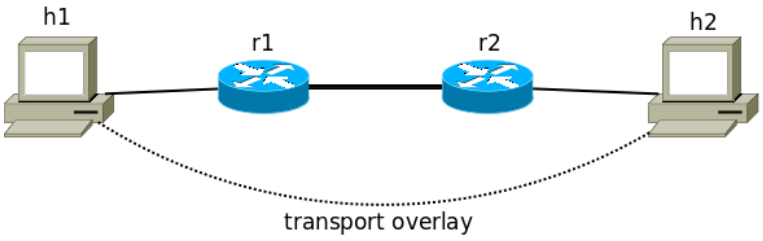

Figure 3.1: Illustration of our test setup

To simulate the behaviour of our applications in a real network, we built a mininet-based environment [35]. It emulates two end hosts connected to two routers as depicted in Figure 3.1 with the links, the IPs addresses and the routing tables configured manually. This simple network constitutes our basis to measure the native performance of QUIC (chapter 3.3) and our applications (chapters 5 and 6). We wrote python scripts to launch our applications, test their connectivity and measure performance. Those scripts allow to reproduce any experiment described in this thesis. This code is published on Github [1] and is publicly available to allow

[1] `https://github.com/rfloriot/New_applications_above_QUIC`

reproduction of the experiments.

To simulate different network conditions, we opted for the command `tc` [36] which is very flexible. It uses `HTB` and `netem` queuing disciplines on the link r1-r2 to apply losses, delays or to shape the bandwidth.

Those algorithms require queues, that we set depending on the bandwidth according to the formula:

$$\text{queueSize[packets]} = 1.5 \times \text{BW}[packets/sec] \times RTT$$

This allows to buffer in the queue at least one full optimally tuned congestion window. The 1.5 factor is used to let a bit of slack. We know it will not work on links with very low bandwidth delay product (BDP). As the queue will probably be too small, any small burst would be dropped. But, we don't explore such networks in the document. Indeed, as our applications connect distant networks together, we consider that they see at least a minimum amount of delay. For us, it also does not make much sense to run a VPN or SSH over a link that has a very low available bandwidth.

3.1.3 Tools

We use several tools to perform experiments and collect measurements. For this thesis, we probe bandwidth with iperf [37]. Concurrent and sequential file transfers use apache benchmark [38], which also measures the time to download some files and provides some statistics. Ditg [39] is our preferred tool for live application testing. Finally, tcpdump [40] helps us to see what is happening on the channel between hosts using our application, for instance for the traffic inside a tunnel. Even though QUIC is a fully encrypted protocol, sniffing the network still allows us to obtain important insight about the clear text data running inside our overlays, as well as to extract global metrics, such as the total UDP bandwidth, ...

3.2 Networks characteristics and experimental design

Based on this infrastructure we can measure the performance of our software. Depending on what we want to observe, we perform two kinds of experiments. Either we could pinpoint the reaction of our application for a specific network instance. Or, with a fixed set of parameters, we also explore their behaviour in a wide range of possible configurations.

3.2.1 Network characteristics

In any case, we restrict network characteristics to values that we could easily observe in a residential ADSL network [33]. Bandwidth is between 1 and 100Mbps, delay goes from 10 to 100 ms and we set uniform jitter from 1 to 10ms and loss between 0 and 2%. If we need to select file sizes, depending on the experiment, we pick them between 8 and 8192 kB. Low BDPs are discarded for the reasons already mentioned above. Since `quic-go` focuses on research purpose, we neither characterise over networks with high bandwidth or delay, such as enterprise optical fiber or satellite, where the implementation could become a bottleneck. Transferring at a high bandwidth consumes a lot of CPU, and require a different hardware. High delays can induce queues, which consume memory.

This made us define "base parameter grid" (Table 3.1), which we use later for all our experiments.

parameter	min	max
bandwidth	1Mbps	100 Mbps
one-way delay	10 ms	100 ms
jitter	1 ms	10 ms
loss	0%	2%
file size (optional)	8kB*	8192 kB*

*depending on the experiment

Table 3.1: Base parameter grid

3.2.2 Experimental design

Whenever we want to explore the full reaction in a set of multiple configurations, we use an experimental design inspired by Latin Hypercube sampling [41] from the python package PyDOE[2]. It groups the search space into a grid depending on the number of experiments and pick random values inside each segment. This helps us to have both a well distributed and random experimentation with a minimum number of iterations.

3.3 Reference experimentation

Before describing our programs, we need to outline the basic efficiency of our QUIC implementation. We use it later to have an upper limit on our application performance. To achieve it, we compare QUIC and TCP for file transfers.

3.3.1 Setup

To perform file transfers with QUIC, we conceived a tool called `quic-bench`. It is a simplified clone of apache benchmark. It queries a server to download a configurable amount of data with one or multiple QUIC streams and measures the total transmission time.

We use the same mininet setup, as described in Table 3.1, to transfer several files of different sizes between hosts h_1 and h_2, and compare the time required by `quic-bench` and apache benchmark.

File size

We quickly noticed that `quic-go` was slower than TCP in many cases and thus, we decided to review our parameters.

QUIC and TCP do not require the same number of RTT to transfer data, especially with very small files. This is related to the fact than when QUIC establish a session for the first time, it needs to exchange additional cryptographic information with the server, inducing at least three rounds of CHLO-REJ. In a normal QUIC session, the last two exchanges will be able to carry data in an encrypted, however, non forward secure way. But, unfortunately, this is not currently supported in `quic-go`.

That is why in this reference experimentation, we only download large files, as we need to open a new session each time we perform an experiment. We will relax this requirement later, when measuring our QuicVPN and QuicSSH. As both tools use persistent sessions, they are not impacted by this problem.

[2] `https://github.com/tisimst/pyDOE`

Jitter

But, even with large files, `quic-go` is still often slower than TCP. We observed that reordering is not very well handled with `quic-go`. This could give an advantage to TCP on all further experiments only due to our choice of implementation.

We can measure it by dividing QUIC transfer time by TCP value under different conditions. A number larger than 1 indicates that TCP is faster than QUIC while values smaller than 1 stipulate the opposite. Figure 3.2 illustrates this for the transfer of fifty 256kB files, over a 50Mbps, 1% of loss 50 ms of delay network with jitter from 0 to 30ms. Here, we can see that the ratio between QUIC and TCP grows with the jitter. This indicates that jitter is handled in a less efficient way in `quic-go` than in TCP.

This led us to build a new parameter grid for our experimental design, without taking jitter into account (Table 3.2).

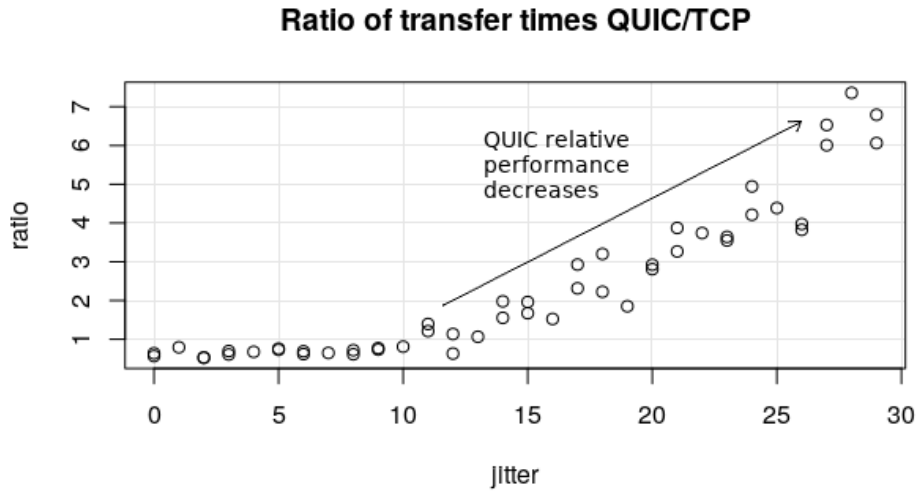

Figure 3.2: Illustration of jitter impact on `quic-go`

parameter	min	max
bandwidth	1Mbps	100 Mbps
one-way delay	10 ms	100 ms
jitter	**0 ms**	**0 ms**
loss	0%	2%
file size (optional)	**1024 kB***	**8192 kB***

*only in the reference experimentation

Table 3.2: Reviewed parameter grid

3.3.2 Single stream

Now that we have a corrected set of parameters, we can begin exploring the performance of QUIC in different conditions.

To see how it behaves with a single stream, we download sequential batches of five 1MB files (in 500 different network conditions from the reviewed parameter grid). Then, we report for each one the median time needed by QUIC to download a file and compare it with TCP using a ratio. Again, a value smaller than 1 indicates that QUIC is faster than TCP. Finally, we compile all the results in a cumulative diagram.

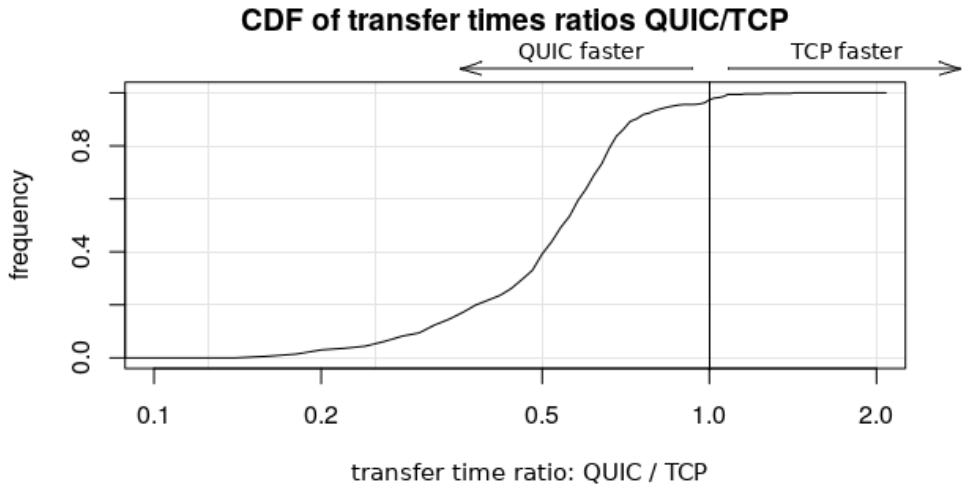

Figure 3.3: Comparison of TCP and QUIC performance

Figure 3.3 illustrates that QUIC is faster than TCP on more than 95% of the iterations. On 40% of the cases, QUIC is even more than twice faster than TCP. The smallest differences appear on links with low losses. In that case, TCP performance is very close to the physical bandwidth, that QUIC cannot overcome.

This comes from the better ability from QUIC to get a higher bandwidth on lossy links. TCP's maximum throughput decreases very quickly [42] its congestion window when it experiences random losses. QUIC's specific retransmission strategies and more aggressive congestion window growth seem to have a positive impact on those situations.

We confirm this by observing the transfer rate when fixing the link bandwidth and delay (here, to 50Mbps and 50ms), by downloading 4MB files, and repeating the experiment 50 times with losses varying between 0 and 2% as shown in Figure 3.4).

Figure 3.4: Link between measured applicative bandwidth and loss rate (single stream)

Here, we can see that for the same loss rate, QUIC is able to get a better application level throughput.

3.3.3 Multi-stream

We repeated the same experiment by adding concurrency. Instead of using 1 QUIC stream, we set QuicBench using 5. And, we compare it to apache benchmark configured with 5 concurrent TCP sessions to download 1MB files.

Here, the results are different. QUIC is faster than TCP only in 20% of the cases and is up to three times slower in some situations (Figure 3.5). Furthermore, it also seems to be related with the loss rate.

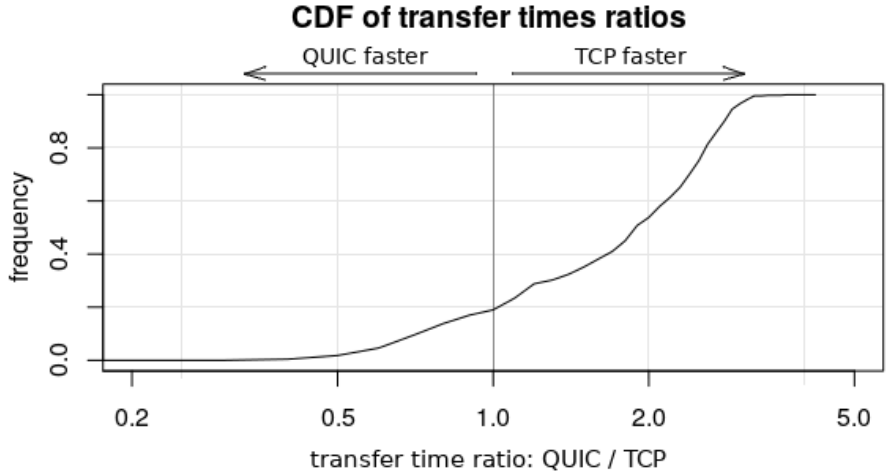

Figure 3.5: Comparison of TCP and QUIC performance (with concurrency)

If we focus on the impact of losses (using 50 downloads, under fixed 50Mbps, 50ms delay network with variable loss rate and again, 1MB files), we can remark that QUIC gets higher transfer rate, but only when losses are rare (figure 3.6). The difference seems smaller than in single stream and it inverts after 0.5%.

Figure 3.6: Link between measured applicative bandwidth and loss rate (with concurrency)

Explanation of the problem

In TCP, losses only impact the particular session that experienced them. But, as QUIC only use one congestion control for all its streams, packet losses impact negatively the overall transfer rate (figure 3.7).

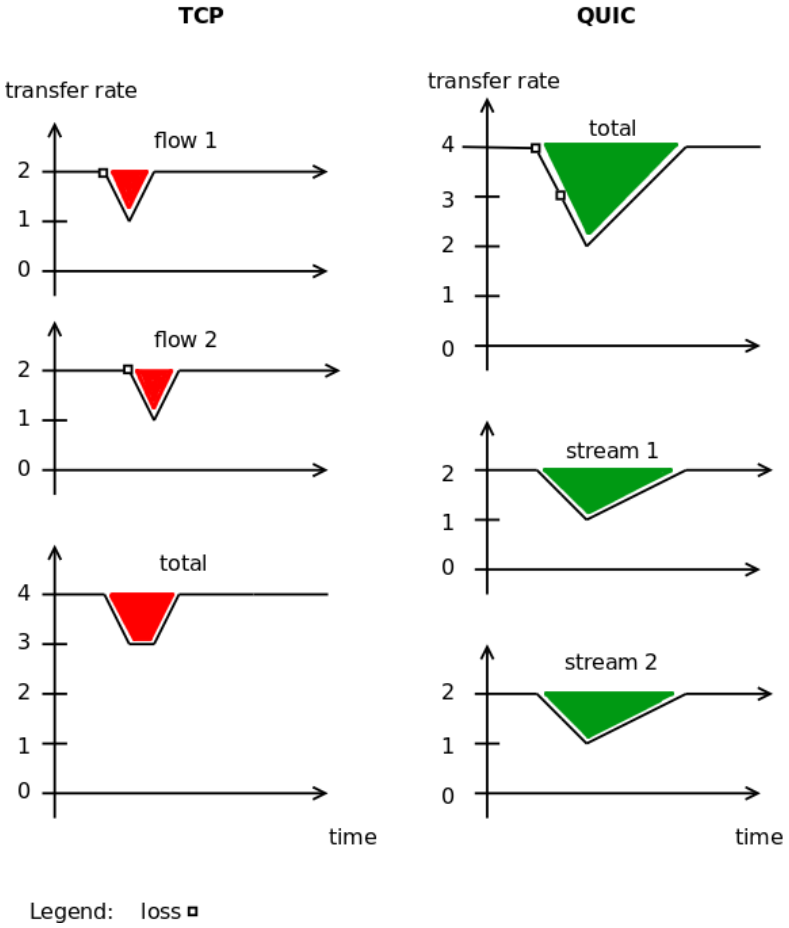

Figure 3.7: Impact of losses on congestion control

If we have a link with X% of losses, N TCP streams act as N independent flows. Each of them experiences X/N% of losses, allowing them to get a higher bandwidth. But, QUIC uses one overall congestion control. It thus reacts exactly as a single TCP session.

If we run the download test either using QUIC or TCP in a 50Mbps, 50ms delay, 1% of loss network, but by changing the number of streams, we can see that the more we have streams, the less QUIC competes with multiple concurrent TCP flows (Table 3.3):

number of streams / TCP flows	file size	total QUIC time	total TCP time
1	8000kB	38s	55s
2	4000kB	38s	30s
4	2000kB	41s	15s
8	1000kB	42s	8s
16	500kB	43s	4s
32	250kB	31s	4s

Table 3.3: Download time using QUIC or TCP with different concurrency level

QUIC built-in solution

Google was aware of this problem. To solve it, they tuned the QUIC congestion control to act as an ensemble of N TCP sessions [18] [43] using an approach inspired by MulTCP [44]. Its basic principle is to grow the congestion window each RTT by $N/cwnd$ and to decrease it when experimenting losses by $(N - 0.5)/N$

The problem for our applications is the constant, default value of $N = 2$. For this kind of benchmarks, as well as for our applications, this is not the correct value. Initially designed for YouTube, it makes sense for QUIC to use 2 streams, one for audio and one for video. We think that not all programs should compete with only two concurrent TCP sessions. It should be adjustable and allow to control the number of emulated connections. Our benchmark, as well as the VPN and SSH tools needs it to be exactly equal to the number of streams opened . This is a requirement to compete with native transfers.

Unfortunately, this is not possible with the current QUIC implementations. MulTCP becomes unstable and does not to improve efficiency as N grows [45]. Moreover, it is often too aggressive and unfair. So we cannot set it to a high value.

If we repeat our experiment of Table 3.3 with N equals to the number of streams, we can observe this in `quic-go` (Table 3.4).

number of streams / TCP flows	file size	total QUIC time	total TCP time
1	8000kB	64s	55s
2	4000kB	35s	30s
4	2000kB	20s	15s
8	1000kB	15s	8s
16	500kB	8s	4s
32	250kB	8s	4s

Table 3.4: Download time of QUIC and TCP with different concurrency level (using N = # streams)

After 4 streams, the ratio performance of QUIC compared to TCP quickly degrades. If we repeat our multi-streamed experience to observe the impact of using $N = \#streams$ (Figure 3.8) on a large set of networks settings, QUIC is still on 50% of the cases more than 50% slower than TCP, as with $N = 2$.

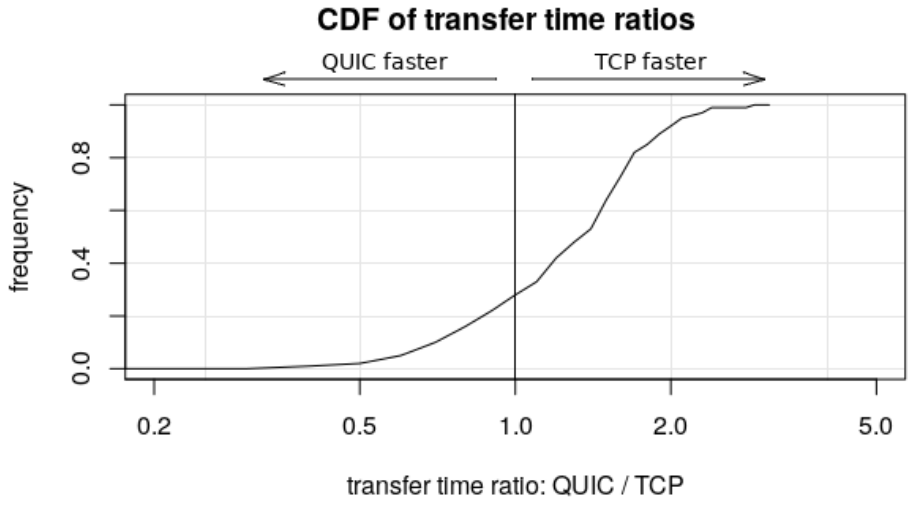

Figure 3.8: Comparison of TCP and QUIC performance (with concurrency) using N = # streams

That is why we will restrict this study to use default value of 2, already experimented in large scale by Google, knowing its aggressiveness compared to a single session and its default with multiple streams.

How to solve it?

To get a comparable performance between concurrent TCP flows and a QUIC session using multiple streams, we need a better congestion control. It must be able to emulate an arbitrarily high number of TCP sessions, and use it in our applications. Probe aided MulTCP [46] can be a good solution to improve this. Non-loss-based congestion controls, such as BBR can also play a role.

But even with those algorithms, allowing applications to set their fairness compared to TCP is still a dangerous choice. It would lead each software to tune it to its maximum value, to get most of the bandwidth in the available links. This problem is a generic concern of QUIC, due to its user space nature and easy customisation. It needs to be solved to allow large scale deployment of new applications.

3.4 Summary

This section helped us to define a strong experimental methodology that we will use to characterize our applications. By applying it on a sample program called "quic-bench", we have outlined the basic performance of QUIC, compared to TCP.

We will use the same approach to measure the performances of our QuicSSH and QuicVPN applications. Especially, we will keep the same topology, experimental design procedure and parameter grid (Figure 3.5), which does not contain jitter, for the reasons expressed before, but still includes smaller files, as our QuicVPN and QuicSSH does not re-connect for every small transfer.

We will also keep the experience we have acquired by measuring quic-bench. Especially, the discussion about the strength will often serve as a reference basis to explain why our programs are faster or slower than TCP with one or multiple streams.

parameter	**min**	**max**
bandwidth	1Mbps	100 Mbps
one-way delay	10 ms	100 ms
jitter	0 ms	0 ms
loss	0%	2%
file size (optional)	8kB*	8192 kB*

*depending on the experiment

Table 3.5: Final parameter grid

Chapter 4

QUIC extension: Client authentication

As both of our applications: QuicVPN and QuicSSH need to perform strong authentication of the clients to identify them and only allow a restricted set of clients to use our tools, we needed to implement client authentication.

Ideally, this should be ensured in the future during handshake by TLS 1.3 [8]. But, as the primary use case of QUIC was to improve web connections, client authentication was not considered as a primary focus and is not available yet in the QUIC handshake of `quic-go`. That is why we decided to implement it in the applicative layer.

Figure 4.1 describes how we achieve this using client and server certificates. It uses the following notations:

- pk is a private key, PK is a public key
- $S_{pk}(x)$ is the signature of x using the private key pk
- $V_{PK}(x, sign_x)$ is the verification of the signature $sign_x$ of x with the public key PK
- $connID$ is the connection id chosen at connection establishment (added to avoid replay attacks from an authentication made with a different server)

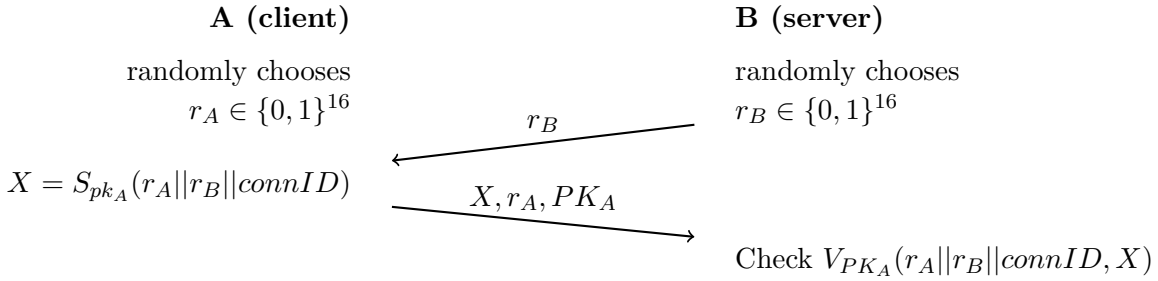

Figure 4.1: Illustration of the challenge-response applicative protocol

In this authentication protocol, the client signs using its private key a random nonce r_A concatenated to a random nonce r_B chosen by the server concatenated to the connection id. Then, it sends the signature, along with the public key to transfer to the server, which will check the received data. This unidirectional authentication is inspired by ISO 9798-3 protocol number 2.

Chapter 5

Application 1: QuicSSH

In this chapter we present QuicSSH. First, we describe what is this program and we explain how it is implemented. Then we give some experimental results by comparing performances between QuicSSH and alternatives running over TCP like SSH and SSF [47]. Finally, as an extension, we introduce the possibility to allow QuicSSH to use multiple paths with Multipath QUIC [48] and we measure the performance gain associated with this new protocol.

5.1 Program description and implementation

QuicSSH is a tool for port forwarding and remote login. It is a functional equivalent of SSH implemented on top of the QUIC protocol rather than TCP. This application benefits from encrypted communications provided by QUIC and makes usage of public and private key pairs for both client and server authentication.

QuicSSH is implemented in the go language for all the components, with 1500 lines of code (comments and blank lines included). It is built on top of the `quic-go` library [7], which provides the QUIC interface. It is made of 3 main components: connection establishment, remote login and port forwarding. All those come with unit tests.

In this section, we explain briefly, for each of those 3 parts, the program behaviour and give some implementation details.

5.1.1 Connection establishment

Before using the remote login or port forwarding functionalities, QuicSSH must set up a channel ensuring confidentiality, authentication and integrity. Most of this work is already present inside the QUIC protocol which ensures reliable transport and encrypted communications. But, any SSH tool must also authenticate the server to see if the client can trust this server. And, it also needs to authenticate the client so that the server can check if it needs to accept or reject this client.

In QuicSSH, client and server authentication are inspired from the SSH keys mechanism presented in section 2.2. It uses public-key cryptography. Both server and client have their own public and private keys. Similarly to SSH, each QuicSSH server stores a list of public keys corresponding to the allowed clients. Also, each client stores a list of public keys (associated to a server IP and port) corresponding to trusted servers. We have decided to not implement the password authentication in QuicSSH as we consider it less secure than the key authentication.

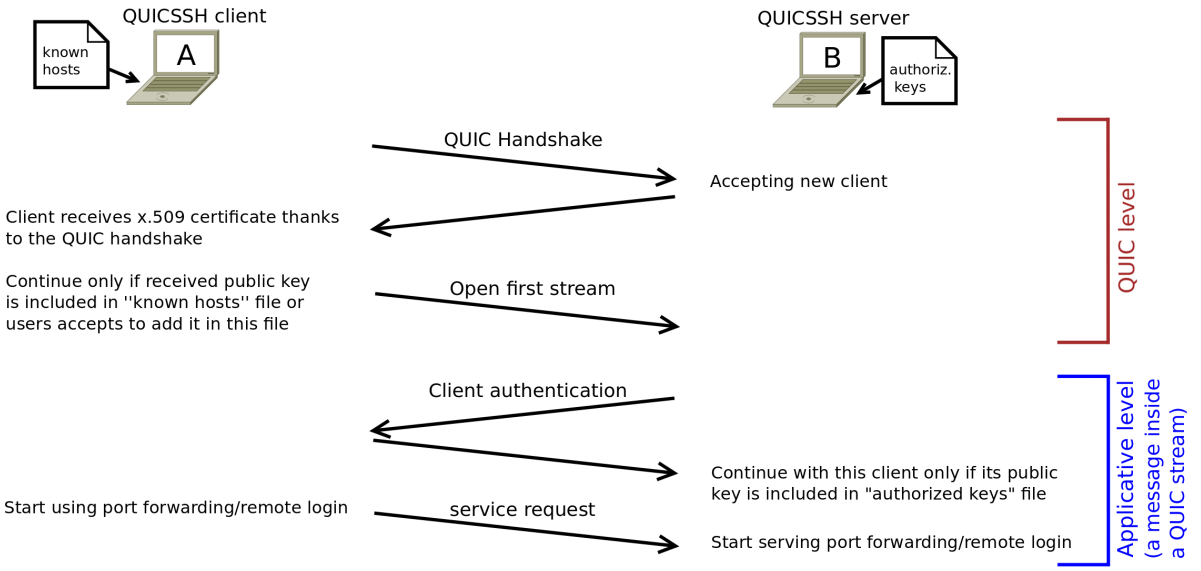

Figure 5.1: Connection establishment in QuicSSH

The Figure 5.1 summarises how connection establishment is implemented in QuicSSH. Each server has a list of allowed clients. We store this list in a file that we call "`authorized_keys`". Similarly, all clients have a list of server public keys, server IPs and server ports, stored in a file that we call `"known_hosts"`.

When a client wants to connect to a server, it first opens a session with the server by performing a QUIC handshake. From this session, it obtains the x509 server certificate and it extracts its public key. As QUIC uses certificates [8], clients are able to verify the certificate chain. But some users may still use self-signed certificates on the machines to which they want to connect to avoid paying certificate registration authorities. In this case the user must say that he trusts the received key the first time he connects to this server. When the QuicSSH client approves the key of the server, it opens a QUIC stream on the established session.

Once the client has verified the server's public key, it needs to send its own public key to let the server authenticate this client. As we assume we won't register all the individual users to certificate registration authorities, we do not use certificates for client authentication. Once the server receives the client public key, it only checks if this key is present in the "authorized_keys" file. If this is not the case, the connection is stopped and the client cannot access the server. But, as `quic-go` does not support transmitting client keys in the handshake (the developers designed QUIC first for HTTP needs), QuicSSH transmits the client's key at the applicative layer on a QUIC stream. Moreover, in order to avoid replay attacks, we need to set up an authentication protocol to let the server trust the key it receives. We achieve it by using our secure protocol defined in chapter 4.

After that, to finish the connection establishment, the client simply sends a 1-byte service request message on the first stream that was opened to indicate which mode to run for this client. The mode can be either port forwarding, remote login or both. This request message only contains one value: 0x01 for remote login, 0x02 for port forwarding or 0x03 for both modes.

5.1.2 Remote login

As described in section 2.2.3, remote login allows a user to execute remotely commands in a terminal window. In our program, the commands and the outputs are sent through a QUIC stream and are thus encrypted.

This part is launched only if requested at service selection. We have implemented the remote

login inside QuicSSH by building it on top of Telnet. The server launches a Telnet server[1], accessible through TCP. When a client asks for opening a new QuicSSH interactive session, our server application calls Telnet in localhost and pipes Stdin and Stdout towards the QUIC stream. In this way the end host running QuicSSH client gets all the functionalities provided by Telnet. This also provides a simple way to ask clients to authenticate inside the server by giving a username and password. (Public key cryptography in QuicSSH implementation only authenticates the client machine but not the user directly, this is why password is still asked). We dedicate a stream per interactive session in QuicSSH.

From a security point of view, Telnet is not a secure remote access protocol as it uses unencrypted communications [49]. But here considering we only need it on the local interface, if either we block connections from the internet towards the Telnet server or if we configure Telnet to accept only local connections, this becomes reasonably secure. What middle-boxes see is just encrypted traffic running on top of UDP (QUIC).

5.1.3 Port forwarding

The port forwarding tool, as described in section 2.2.3, is an application that redirects connections between a client and a server. It achieves this by going through two intermediate hosts that forwards the received payloads. In QuicSSH, we establish the QUIC session between those two intermediate hosts to ensure secure and reliable forwarding.

If a client asks for port forwarding service, the server gets ready to accept new streams on the session shared with the client. One QUIC stream is opened for each TCP connection that uses the port forwarding. For example, if content from port 1234 in localhost is forwarded to a second machine on port 80 with QuicSSH, each client performing `wget` on localhost:1234 will have its own associated stream. This choice stands for simplicity and will be useful to prevent head of line blocking configuration where a single client failure or packet loss affects all the forwarded connections.

QuicSSH supports two kinds of port forwarding: local port forwarding and remote port forwarding. In local port forwarding, the QuicSSH client asks to forward any connection ending locally on a given port towards the QuicSSH server. This connection will then be finally forwarded to a given address and port. For example, the command `-L 1234:localhost:80` relays any connections on client machine arriving locally at port 1234 to a QuicSSH server that will forward it locally to port 80. Remote port forwarding is similar but in this situation the QuicSSH client asks to the QuicSSH server to forward connections the server receives on a specific port towards the client. This client will as its turn forward it towards a given address and port.

As it is the case with SSH, we only implemented TCP port forwarding. We could adapt the program to also support UDP. But this would be slightly more complicated than TCP as, in order to support concurrent messages, we need to keep an explicit mapping on the QuicSSH client between the source (IP + port) and the stream to use. Similarly on the server, an explicit mapping would be necessary between the stream ID and the port used as source to contact the destination of the port forwarding. All this is more implicit with TCP, as we associate a stream ID to each new connection.

[1]The Telnet server is launched in the foreground and stopped when the QuicSSH server is stopped. Also the port used by Telnet is not the default one in order to try to avoid bots doing root login.

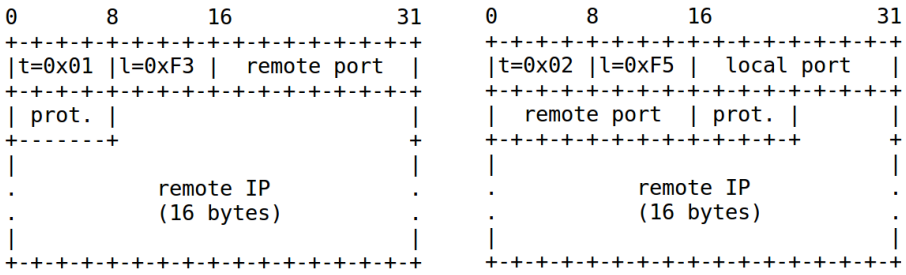

Figure 5.2: Control messages for local port forwarding (left) and remote port forwarding (right)

When a client asks for port forwarding, it needs to inform the server of some parameters. We created control messages to let the client give to the server all information it needs. This is presented in the Figure 5.2. The control message at left carries information for local port forwarding while the one at right contains fields used for the remote port forwarding. Those are both sent at applicative level (When opening a new stream for port forwarding in QuicSSH, this is the first message sent on it). They both use a *type-length-value* encoding (TLV). Here is the description of each field:

- `t`: this is the type byte, it can be 0x01 for local port forwarding or 0x02 for remote port forwarding.
- `l`: This is the length field, it is fixed and it depends on the type of port forwarding.
- `local port`: This field gives the local port used for port forwarding. Having this field encoded on 2 bytes means that port must be in the range [0 - 65535].
- `remote port`: This is the remote port used for port forwarding, encoded on 2 bytes.
- `prot.`: The protocol field encodes the protocol number of the forwarded message. QuicSSH only supports TCP but this field is there to facilitate the extensibility of the application.
- `remote ip`: This field gives the final destination of port forwarding. Encoded as IPv6 (16 bytes), it can handle IPv4 too by using a fixed IPv6 prefix : 64:ff9b::/96 (RFC 6052).

Figure 5.3 summarises in detail the process implemented for local port forwarding. To intercept connections arriving on a given port on QuicSSH client, we use a socket (a go listener) to listen to new connections. When such new connection arrives, we open a new QUIC stream and the first message that is sent on it is the control message defined above. Port forwarding only transmits payload data. This means that our program stops connections between the port forwarding client and the final destination, on an intermediate node (host A in the Figure 5.3) and then reopens it on another intermediate node (here B). QuicSSH uses a dedicated QUIC stream to send the application data between those intermediate nodes.

When the socket listener obtains end of file message or when the TCP destination closes the connection, we close the stream to announce it to the remote QuicSSH host. Finally, when a host sees that the stream is closed, it closes the associated network connection if not already done.

Implementation of remote port forwarding is very similar. It only reuses components of local port forwarding implementation. The QuicSSH server plays the role of the first port forwarding intermediate by opening the socket. And the QuicSSH client acts as the second port forwarding intermediate. It reads control messages and opens connections to final forwarding destination to transfer payloads.

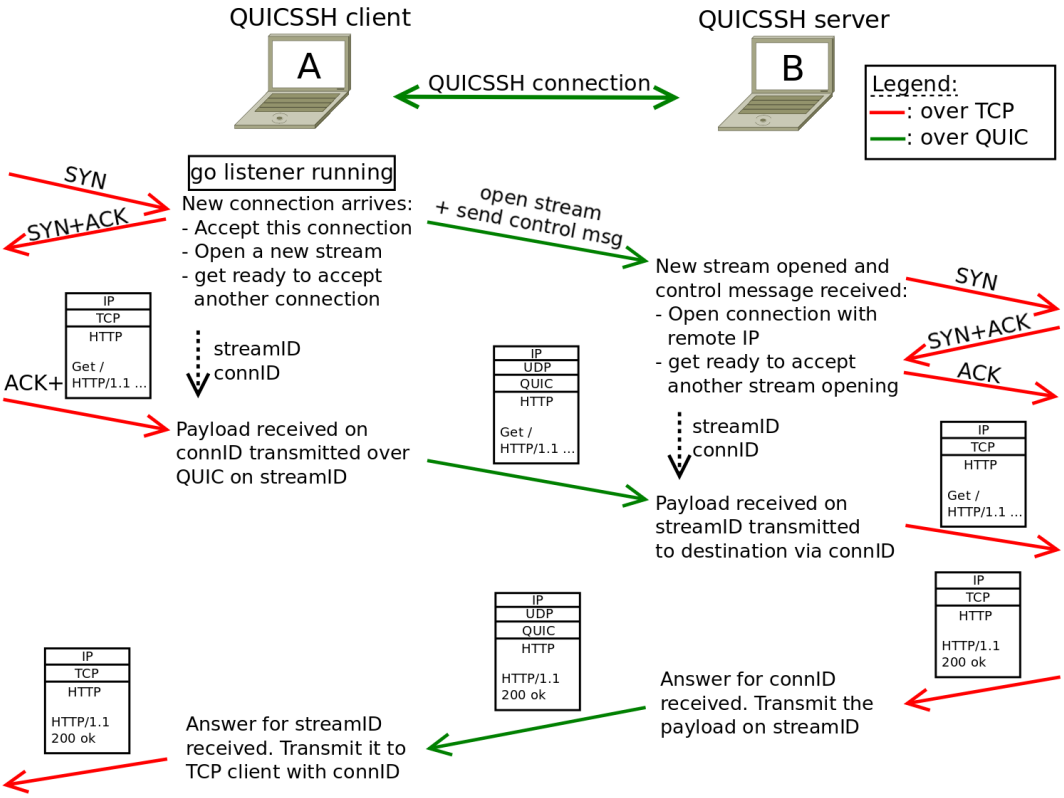

Figure 5.3: Local port forwarding implementation

5.2 Security

We now compare QuicSSH with the standard SSH protocol [20] in terms of security. Confidentiality, authentication and integrity are explored to make the comparison.

5.2.1 Confidentiality

First we consider confidentiality. Both QuicSSH and SSH use encrypted communications with several ciphers available like 3DES, AES-128, AES-192 and AES-256 [50][21]. We can thus choose a proper key length that suits our needs (trade-off between security level with the key size and encryption/decryption time) in those two protocols. Concerning key exchange, Quic supports both pre-shared keys (in 0-RTT mode) and Diffie-Hellman key exchange while SSH uses only the Diffie-Hellman key exchange protocol. There is thus not a weaker protocol in terms of confidentiality as several encryption algorithms are available. The strength of encryption relies mainly on the user that needs to make a proper choice for the cipher method.

5.2.2 Authentication

Then, concerning authentication, both QuicSSH and SSH [23] use public key cryptography to demonstrate the identity of the parties. SSH also offers the password authentication method. We consider the password method less secure as a password is generally smaller (in bits length) than a RSA private key (which is at least 1024 bits length). Also the generation of private keys uses randomness, which is often not the case for the passwords (risk to be vulnerable to dictionary attacks). We thus only consider authentication with public key cryptography. In QuicSSH, the client authenticates the server at QUIC handshake phase. But concerning the client, we perform its authentication at the applicative layer. It uses the (secure) authentication protocol introduced in the Section 4. In the future, client authentication should be implemented in Quic

handshake thanks to TLS 1.3 but it is not yet the case in the `quic-go` library we use. In SSH, authentication is performed on top of the "encrypted layer". It allows several authentication protocols.

In both QuicSSH and SSH, the client stores a file with known public keys of servers. Similarly, the server stores in a file the list allowed public keys of clients. If the server receives a connection from an unknown client, it simply refuses it. This is less simple for the client. The first time a client connects to a server, it does not already know the server's public key. In SSH there is so potentially a risk of man-in-the-middle attack at the first connection [20] (someone could intercept the connection request and reply with its own key). In QuicSSH, as x.509 certificates are available, the server certificate can be signed and the client can thus verify the certificate chain. But it could be simpler for end users to avoid using signed certificate so we did not oblige QuicSSH client to verify the certificate chain (it is optional). QuicSSH is thus slightly better in terms of authentication if we consider that the server uses a certificate signed by a certification authority (and that the client effectively verifies it). Otherwise, if a the server uses a simple self-signed certificate, there is then no stronger protocol between QuicSSH and SSH from an authentication point of view (considering SSH uses public key cryptography rather than simple password authentication).

5.2.3 Integrity

SSH ensures the integrity of each packet by adding a MAC that is computed on the packet content, the sequence number and, of course, the shared session key [21]. This is the same for QUIC with either QUIC Crypto or TLS: it uses a keyed MAC to ensure integrity [50].

5.2.4 Usage of Telnet inside QuicSSH

Not to reinvent the wheel, we decided to use Telnet to build our remote login service, as it does all we need. Telnet is not secure as it uses clear text communications but here as we use it on top of our channel that provides confidentiality, authentication and integrity, this is secure. Also we configured Telnet to run in the foreground, only when QuicSSH server is launched and we changed the port used for Telnet to try avoiding brute-force attacks on standard Telnet port. An additional security layer can be set-up by adding a firewall rule telling that this new port of Telnet can only be accessed locally and not from the internet.

5.2.5 Conclusion on security

By looking at security, confidentiality and authentication, we can say that QuicSSH is as secure as SSH; or even better than SSH if the QuicSSH server uses a certificate signed by certification authority. The usage of Telnet is also not a problem because we use it on top of a secure layer, it is properly configured and we can even add firewall rules to discard external connections.

5.3 Influence of the number of acknowledgements

Before performing the experiments of this chapter, we remarked that our default version of `quic-go` (0.7) was inducing increased delays in the port forwarding. We discovered that the main reason was the new frequency of QUIC ACK frames. In version 0.7, QUIC sends 1 ACK every 10 packets, while in the previous version, it was 1 every 2 packets, as in TCP. This choice comes from the adoption of a new delayed ACK strategy [51] in QUIC, which, among others, increases the delayed ack threshold and use a different timer computation. While it can be good to reduce the number of packets sent, it can also harm reactive applications, as QuicSSH.

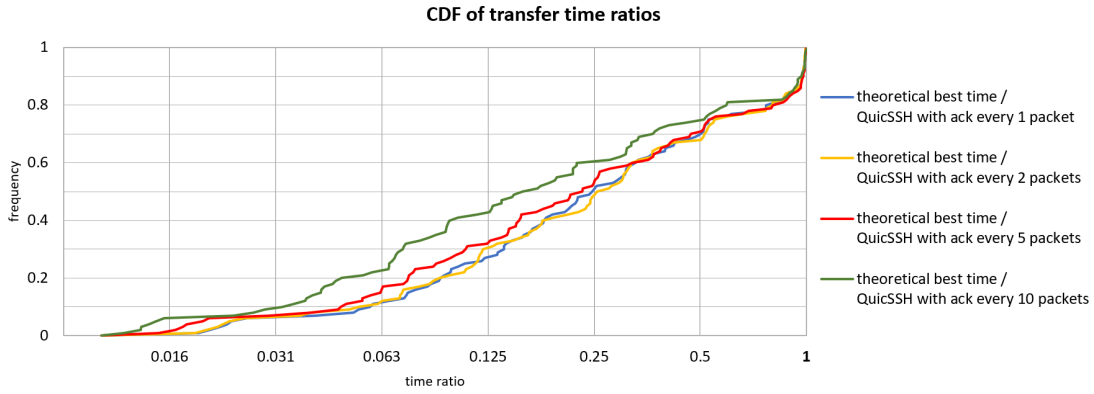

Figure 5.4: download times ratios CDF

Figure 5.4 shows the impact of the QUIC ACK rate on the performance of QuicSSH. The experiment is performed on our full parameter grid (loss: 0 to 2%, delay: 10 to 100 ms, bandwidth: 1 to 100 Mbit/s and file sizes: 8kB, 64kB, 256kB, 1024kB, 8192kB). We introduce the theoretical best download time in order to present those time ratios. The theoretical best download time (in milliseconds) is a lower bound computed as:

$$\frac{\text{file size(kB)} * 8}{\text{bandwith (kbit/s)}} * 1000 + \text{RTT (ms)}$$

It is indeed a lower bound as it includes only once the RTT time and also because the losses do not impact the result value.

On this graph, we divide the theoretical best time by the time actually measured with QuicSSH. Consequently, the performances of QuicSSH are better when the curve moves vertically towards the bottom of the graph and, conversely, it is less good when the curve moves vertically towards the top of the graph. We thus see clearly that having a ACK frame every 10 packets is less good for performances than having it every 1 or 2 packets received.

Concretely, by using a version with a ACK every 2 packets rather than a ACK every 10 packets, in several situations (the one with a file size equal to 8192kB and a loss rate higher than 1%) we win up to 30 seconds. And we generally at least win 1 second when the size of transfer is greater or equal to 1024kB. There are also some rare cases with very small files where we lose some time, at most 100 milliseconds.

Reducing the number of acknowledgement could be useful if the overhead produced was a bottleneck but this experiment do not lead to this conclusion. We suppose that QUIC is more reactive when using 1 ACK every 2 packets instead of 1 ACK every 10 packet, especially in lossy environment. Maybe having faster feedbacks allows QUIC to be more reactive on losses by getting more quickly duplicate acknowledgement and thus performing faster a “fast retransmit”.

That is why we have downgraded the version of `quic-go` to 0.6. This allows us to have a fairer comparisons with TCP based solutions, which emit one ACK every two packets. But, this choice is made only for the QuicSSH experiments (others are done with the version 0.7). Newer `quic-go` versions still have interesting features, such as pacing. The perfect solution would be to find an algorithm that allows both to reduce the number of ACKs, while keeping the possibility to do fast retransmissions.

5.4 Port forwarding performance

In this section, we compare the port forwarding of QuicSSH with SSH, with the situation where there is no port forwarding and with secured socket funneling (SSF). SSF is a related work for port forwarding, built for performance [47].

We first run some simple tests on specific network conditions. Then, we perform some experimental designs to cover a wide range of parameters. The section about methodology (Section 3) explains the environment and software that we use (mininet, tc) in our experiments. It also introduces the topology and the network characteristics that we vary in the experimental design. The first experimental design of this section gives the results without concurrency and then the second one, with concurrency.

5.4.1 Experiment: downloading files without concurrency: a simple test

We start the experimentations by a simple test to show that QuicSSH works as expected with some specific network parameters. We choose a network with a reasonable bandwidth (10Mbit/s), a low one-way delay (5ms) and without jitter. We then download a file of 1Mo using the different port forwarding tools tested (QuicSSH, SSH, SSF) and also when using TCP directly without port forwarding.

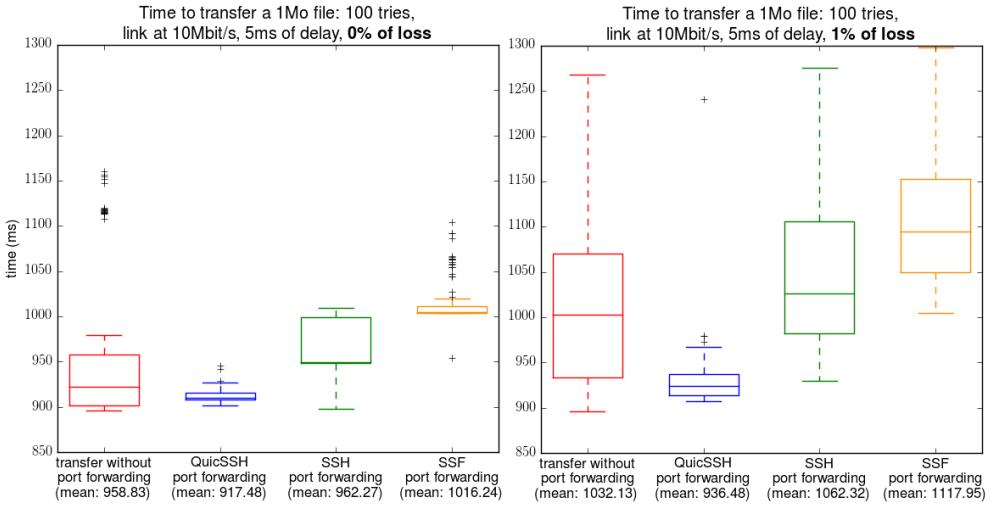

Figure 5.5: Box plots showing download times depending on the forwarding tool to use, without loss (at left) and with 1% of losses (at right)

We make the time measurements 100 times in order to obtain a more accurate mean and to draw a box plot also giving an indication about the median and the variance. Figure 5.5 presents the box plots obtained and each corresponding mean value.

As we can see, in this specific network condition, QuicSSH seems on average faster with losses (1%). It also seems slightly faster without losses. The confidence interval in both cases do not let us conclude that QuicSSH is faster for sure as they overlap (except between QuicSSH and SSF where the confidence interval do not overlap). We can also remark that the variance is lower for QuicSSH compared to the other transfer tools, particularly when there are losses. Also when there are losses, the difference between the mean download times is more marked between QuicSSH and the other download alternatives. This is related to the fact that QUIC handles better the losses as it was already discovered in the Section 3.3.2.

This simple experiment cannot let us make a generalisation about the performances of QuicSSH compared to other port forwarding tools. In the next section we will thus perform an experimental design to cover a wide range of parameters and see the percentage of network conditions in which QuicSSH is faster.

5.4.2 Experiment: downloading files without concurrency: experimental design

This second test tries now to measure the performances over various network conditions using experimental design technique. (The conditions tested are loss: 0 to 2%, delay: 10 to 100 ms, bandwidth: 1 to 100 Mbit/s and file sizes: 8kB, 64kB, 256kB, 1024kB, 8192kB). We use apache benchmark [38] to measure download times in those different conditions. At this step, we do not introduce concurrency. At total, the experiment runs over 200 combinations of file size, delays, losses and bandwidth. We have built a dataset containing the median download time we obtained, over 30 tries, for all these 200 configurations (and for the 4 programs: QuicSSH, SSH, SSF and "no port forwarding").

Fastest solution

A first simple statistic computed on this dataset is the proportion of network conditions in which QuicSSH is the fastest solution. We observe that QuicSSH is the fastest solution in 188 situations over the 200 cases studied. This means that in 94% of the cases, SSH, SSF and "direct transfer" are all three slower than QuicSSH.

Pairwise comparison

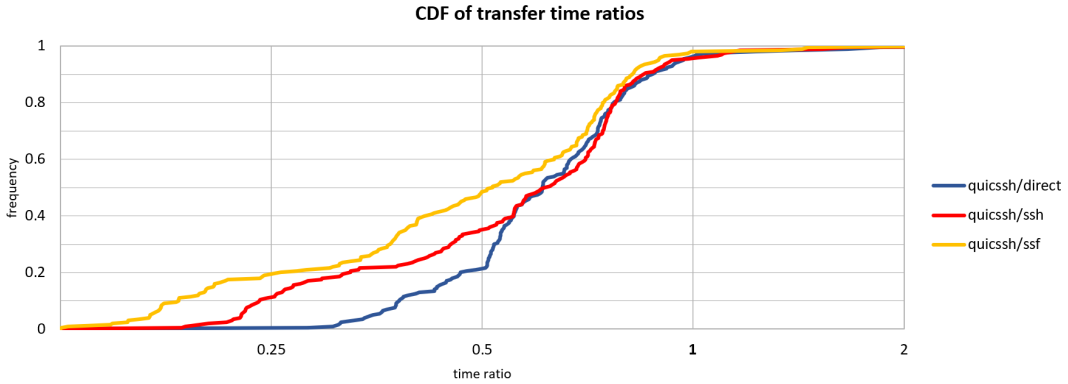

Figure 5.6: download times ratios CDF

A second simple statistic also showing good performances for QuicSSH is a time ratio between QuicSSH and the competitors. As it can be seen on Figure 5.6, QuicSSH is better than SSH in 95.5% of the cases, than `"direct transfer"` (no port forwarding) in 96% and than SSF in 98% of the cases.

Gain measure

To be more precise than just saying QuicSSH is most of the time faster, we can compute the average percentage of time reduction using QuicSSH compared to each other download strategy. On average, we get that QuicSSH is 35.45% faster than a direct transfer, 39.23% faster than SSH and 46.35% faster than SSF.

Similarities between TCP based solutions

After giving those raw statistics, we notice that the time ratios `QuicSSH/direct`, `QuicSSH/SSH` and `QuicSSH/SSF` are all strongly correlated between each other (coefficient of correlation is above 0.7 when comparing all those pairs of ratios for all network conditions). In the next

paragraphs we thus won't detail comparison between QuicSSH and all other transfer strategy individually. We will mainly use SSH for making comparisons.

Influence of the transfer size

We now try to understand in which kind of scenario QuicSSH is faster or slower than competitors. We first take a look at the transfer size. We notice that QuicSSH is faster in 100% of the network conditions tested than SSH when using the shortest file size considered (8 kB). This is also true if we compare QuicSSH with SSF or with a direct transfer over TCP. On the opposite, if we consider all the network conditions tested when file size is equal to 8192kB (the largest file considered), we then notice that the percentage of cases when QuicSSH is better than SSH reduces to 90%.

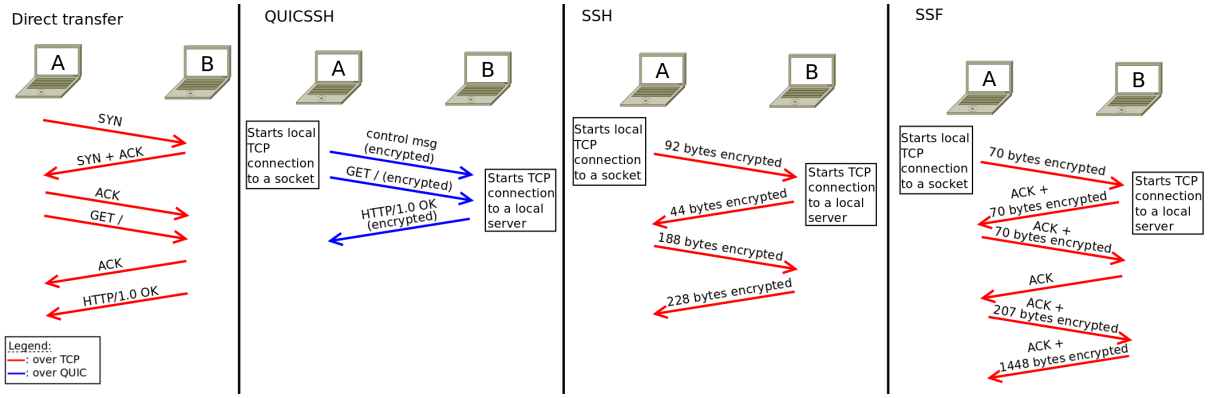

Figure 5.7: Packets observed between two hosts when starting a file transfer between them

Direct transfer	QuicSSH	SSH	SSF
2.01	1.02	2.02	4.79

Table 5.1: Median number of RTT actually elapsed when transferring a single-byte file

We found an explanation about why smaller transfer size induces better QuicSSH performances compared with other solutions. When starting to forward any data, we can see thanks to traces of packets captured that QuicSSH has one round trip time (RTT) less than direct transfer and SSH and even two RTT less than SSF. The Figure 5.7 represents what can be seen inside a trace captured between a source and an HTTP server when downloading a file. The schemas are stopped as soon as we receive a subset of the file. We can see that direct transfer and SSH need both 2RTT. We also remark that SSF needs 3 RTT. QuicSSH, on the opposite, needs only 1RTT for starting receiving actual data.

We can do an experiment to show that QuicSSH has at least one RTT less than the other solutions tested. This experiment consists in measuring the median RTT (a RTT is computed as $transfer\ time/(2 * delay)$) over 100 different scenarios (we let the bandwidth and the delay vary, but we fix the loss to 0%) when downloading a single-byte file. We obtain, for each port forwarding solution, 100 different values for the number of RTT and we get the median. The Table 5.1 presents the median obtained. As we can see, QuicSSH has close to 1RTT as expected and we will need to wait around 2RTT when using SSH or a direct download over TCP. The median value measured for SSF is a little more surprising, this is over our first expectation. When looking at some traces of packets, we clearly remark that some time is lost on the SSF client and on the SSF server. For instance, we see at a time `t` that the server receives a message from the client but it answers only at around `t+40ms` without using network connectivity between those two messages. We cannot understand and explain why this behaviour happens for SSF (There is no paper or clear documentation available online).

This finding shows a good advantage for QuicSSH when considering the internet use cases where many small files (less than 100kB) are exchanged (for example HTML pages, stylesheets, ...) [52].

We see less the gain of having less RTT at start when considering larger files because this gain is something fixed. It does not depend on the file size and thus it has less impact if global download time is larger. (For example, a reduction of 0.1 seconds is more visible if total download time is 0.5 seconds than if this is 10 seconds).

Influence of the delay

We notice a light influence of the delay on the QuicSSH performance compared to TCP based solutions. If we split the dataset in two parts: fewer than 50ms of delay and over 50 ms of delay, we obtain that, in the first part, QuicSSH is better than SSH in 98.8 % of the cases. This decreases to 92.6% in the second part (the one with delays greater than 50ms). The performance gains seem thus slightly better when delays are shorter.

Also regarding delays, we made several experiments to try to discover if being on top of an applicative protocol (QUIC) rather than a kernel optimised transport protocol (TCP) could induce important overhead but we did not notice important differences.

Influence of the bandwidth

QuicSSH shows slightly better results when bandwidth is higher. Indeed, QuicSSH is faster than SSH "only" in 92.1 % of the cases when the bandwidth is at most 20Mbit/s and this grows to 97.5% of the cases when the bandwidth is between 80Mbit/s and 100Mbit/s.

Influence of the loss rate

Finally, concerning losses, as for the bandwidth, results for QuicSSH compared to SSH are less good when loss rate is lower and better when loss rate is higher. Indeed if we consider only the rows of our dataset when loss rate is lower than 0.5%, we notice that QuicSSH is faster than SSH in 86.6% of the cases. When losses are higher than 0.5% then the frequency in which QuicSSH is better than SSH grows to 98.7%.

The performance gains for QuicSSH when loss rate is higher corresponds to what we have found in the Section 3.3 , when comparing QUIC and TCP regarding losses. This is because QUIC handles better losses thanks to a specific retransmission strategy and a more aggressive growth for the congestion window.

5.4.3 Experiment: downloading files with concurrency

The first experiment used always one stream at a time. We now consider a second experiment: allowing several clients to use the port forwarding simultaneously. We launch the same experimental design set-up with a concurrency level equal to 5 (there are 5 simultaneous clients which reflects in 5 dedicated streams on a single session inside QuicSSH).

Figure 5.8 presents the global output of our experiment. We computed the ratios of file download times over several combinations of link characteristics and for several file sizes (see Section 3.3 for the ranges of parameters we use). We can clearly see that there is an important difference between the bottom CDF curve `"quicssh/direct"` and the two other, `"quicssh/ssh"` and `"quicssh/ssf"`.

We first consider QuicSSH compared to direct transfer. We clearly understand that the direct transfer performs generally better than QuicSSH. For example, if we consider the ratio value equal to 1, we see that QuicSSH is faster than direct transfer in only 37.5% of the network conditions tested. This is far from the 96% measured in the single stream version. We can also see by looking at ratio equal to 2 that there is 46.5% ($100\% - 53.5\%$) of the cases where direct

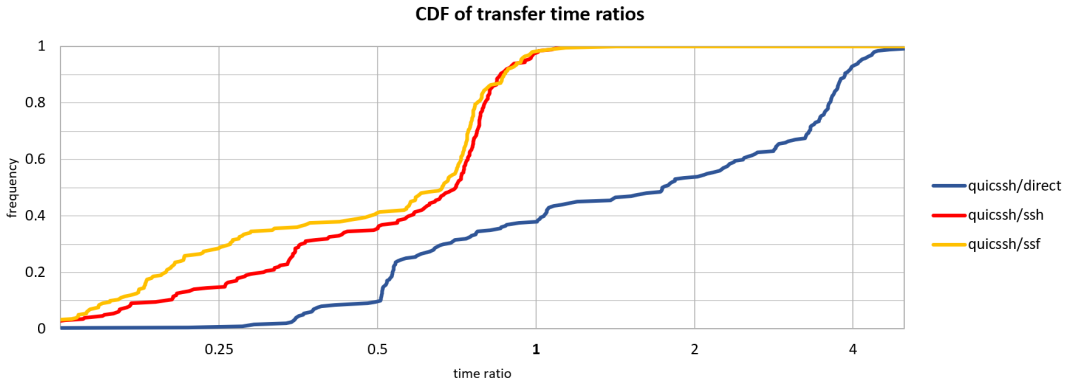

Figure 5.8: download times ratios CDF

transfer is 2 times faster than QuicSSH. On the opposite we can also remark that QuicSSH is 2 times faster than direct transfer in only 9.5% of the network conditions tested.

As stated in Section 3.3.3, QuicSSH is often slower than a direct transfer because QUIC has a single congestion control over all the streams opened. As stated before, in TCP, a loss in one of the concurrent downloads only affects the download rate of this specific session. This is particularly marked when loss rate is higher. As explained in section 3.3.3, a possible solution could be to change the congestion control so that it acts as an ensemble of N TCP sessions by using an approach like MulTCP. But unfortunately, it was shown to be unstable and often too aggressive and unfair. Another solution could be to use a non loss-based congestion control such as BBR.

If we now take a look at the performance ratio `"quicssh/ssh"` and `"quicssh/ssf"`, by considering the ratio equal to 1 on this graph, we obtain that QuicSSH is faster respectively in 97.5% and 98%. This is very close to the single-stream performance. QuicSSH has the advantage, when it uses multiple streams, to reduce head of line blocking compared to SSH and SSF where the concurrent port forwarding are multiplexed over a single session. This is why we obtain good performances with a CDF of performance ratios similar between the single-stream mode and the multi-stream mode.

5.5 QuicSSH over Multipath QUIC

We now test Multipath QUIC [48], an extension of QUIC to provide the capability to use several paths per session. This has the advantage to reduce theoretically the time needed for large transfers by aggregating the bandwidth of the different paths used and to recover faster from path failure compared to single path version.

We compare the performances of QuicSSH over a single path (library `quic-go` [7]) with the performances of QuicSSH over Multipath QUIC (library `mp-quic` [53]). The `mp-quic` version tested is a fork of the `quic-go` library (version 0.6). We thus still use the version 0.6 of `quic-go` in order to perform fair comparisons.

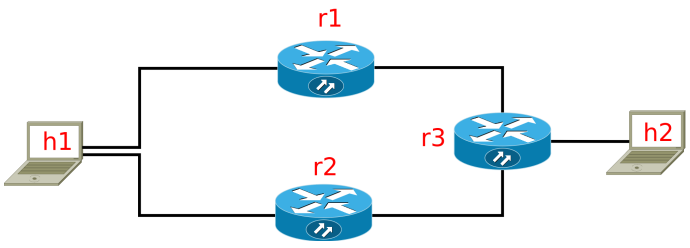

Figure 5.9: Topology used for the experiments with `mp-quic`

In order to perform experiments over multiple links, we need to use a different topology from the one presented in Section 3.1. Figure 5.9 presents this new topology. It is implemented inside a single computer using a mininet simulation. There are 2 paths between client `h1` and server `h2` (`h1` has thus 2 corresponding IP addresses). `r1`, `r2` and `r3` are 3 routers. On the north path, we use `tc` on the link `r1-r3` to simulate the different network characteristics we need. On the south path, we perform the link characteristics simulation on the link `r2-r3`. The links `r1-r3` and `r2-r3` do not have the same characteristics in our experiments. This is because there is generally no reason that 2 paths have the same performances. For instance, the Wi-Fi can be faster than the 4G on a smartphone that is using both networks at the same time.

Parameter	minimum	maximum
Bandwidth	1 Mbit/s	100 Mbit/s
Delay	1 ms	25 ms
Losses*	0%	2 %
File sizes	8kB	8192kB

* if losses are enabled.

Table 5.2: Parameters used for the experimental design in the section using Multipath QUIC

As suggested in the paper about Multipath QUIC [48], We run an experimental design with and without losses. Table 5.2 presents the range of parameters we consider. Using those ranges of parameters allows to compare over several kinds of network conditions the multipath version of QuicSSH with the single path version of this same port forwarding tool. In those two experimental design (without losses and then with losses), we tested 200 different network characteristics (with each time a network characteristics on link `r1-r2` chosen independently from the one of the link `r2-r3`). Each measure reported is the median value obtained over 30 independent tests using the same parameters for the network.

5.5.1 Experiment without losses

Figure 5.10 presents the cumulative distribution function of the transfer time ratios between QuicSSH multipath and QuicSSH single path. We construct it with all the median transfer time obtained with the experimental design. It is important to notice that for the single path version, we always consider the path that provides the best result. We should thus expect that QuicSSH over `quic-mp` performs as fast as QuicSSH over a single path. It can eventually be faster if the second link (the one that provides slower results in single path mode) is still useful compared to the performance of the first link. (For instance, if link 1 has 100Mbit/s and link 2 has 50Mbits/s then quic-mp may benefit from the second link, which is maybe not so relevant if the second link has only 1Mbit/s).

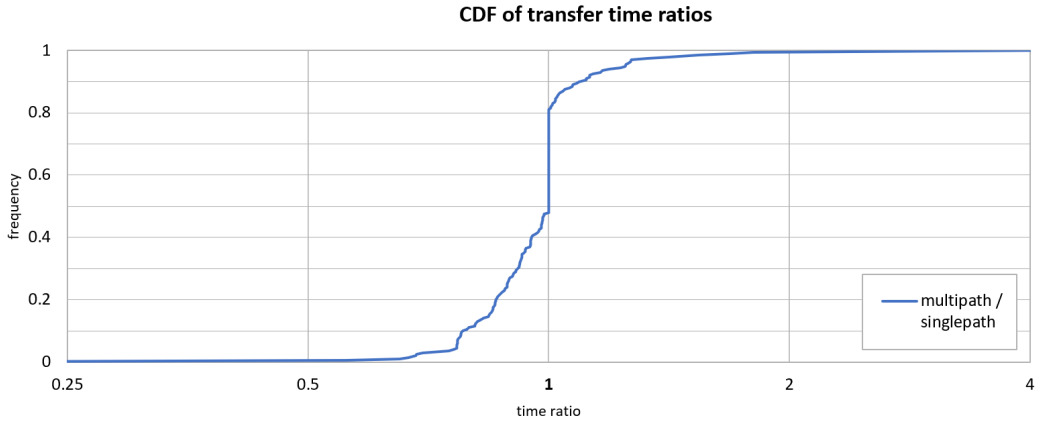

Figure 5.10: download times ratios CDF

On Figure 5.10, we can consider the position "$ratio = 1$". It reveals that QuicSSH with Multipath QUIC is as fast or faster than QuicSSH over a single path in 81% of the network conditions tested. We also remark that QuicSSH over multipath is nearly never more than 2 times slower than QuicSSH in single path (position "$ratio = 2$" on the graph). Indeed only 0.5% of the experiments show a ratio higher to 2. On the other hand, we never see a ratio lower or equal to 0.5 in the 200 network conditions tested. We thus did not remark a condition where QuicSSH using multiple paths is 2 times faster (or more) than over a single path (but we did not tried file sizes larger than 8192kB in our experimental design).

We also observe that there are around 33% of the experiments, in this experimental design, where the ratio is equal to 1. This corresponds to the shortest files transferred (8kB and 64kB). Multipath QUIC duplicates the packets over several paths at start when it has not already measured round-trip times for each path [48]. Consequently, the transfer time is the same between single and multipath as both tools use the best path first in both situations (even if a duplicate packet is sent over the other path in multipath).

The performance of QuicSSH over multipath compared to the same tool over a single path increases with the size of the files transferred. QuicSSH over multipath is faster in 85% if we consider files larger or equal to 128kB; it is faster in 87.5% if we consider files larger or equal to 1024kB and finally it is faster in 100% of the situations tested if the file size is 8192kB. We explain this by the fact that the benefit of aggregating bandwidth is more visible with larger files than for smaller files.

Another thing we observe is that if both links have very good bandwidth, QuicSSH over multiple paths performs better than QuicSSH over a single path. We can see it by forcing both links to have a bandwidth higher or equal to 20 Mbit/s. For the file sizes larger or equal to 1024kB, we increase the situations in which QuicSSH over multipath is faster or as fast from 87.5% to 97.7%.

On the other hand, when there is one "good link" and one "bad link" used in the same scenario, the Multipath QUIC performances seem to decrease. Indeed, we can consider a first link having a bandwidth higher than 20Mbit/s and a delay lower than 10 ms and then a second link having a bandwidth lower than 20Mbit/s and a delay higher than 10ms. Then, for the file sizes larger or equal to 1024kB, the frequency of situations in which QuicSSH over Multipath QUIC is faster decreases to 66%. By looking at the Multipath QUIC implementation, we can see that the scheduler module uses first only the path having the lowest RTT. When the congestion window becomes saturated, it then begins to send over the second path having the best RTT. But here the second path can have significantly lower performances thus maybe the path scheduler should avoid using it. Then Multipath QUIC would become as fast as QUIC single path, with the advantage to still recover faster to link failure.

5.5.2 Experiment with random losses

We now consider an experimental design with the same range of parameters but here we add random losses with a rate between 0% and 2%. Figure 5.11 shows the cumulative distribution function of the transfer time ratios for all the 200 network conditions tested.

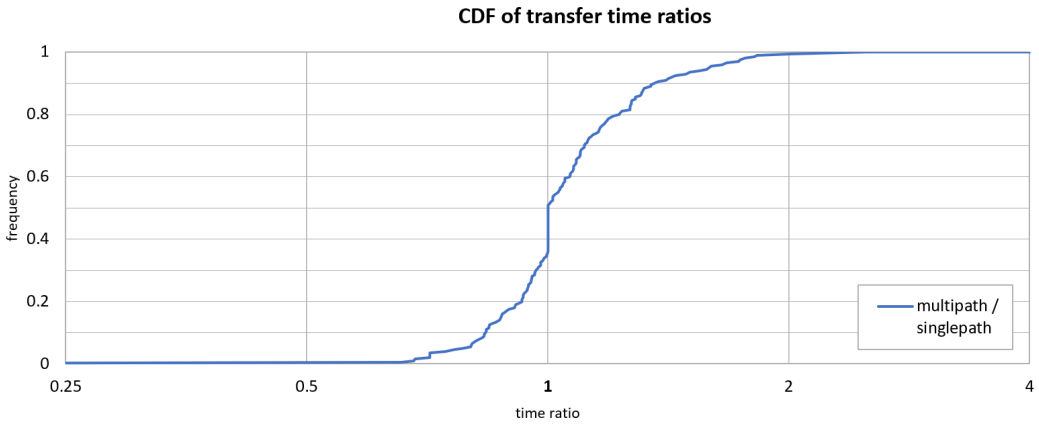

Figure 5.11: download times ratios CDF

As we can see, the curve reveals a decrease of performance for Multipath QUIC on lossy networks compared to the networks without loss. Now QuicSSH over Multipath QUIC is faster or as fast as QuicSSH over a single path in 51% of the cases.

Something interesting here is that the percentage in which QuicSSH over multipath is faster than QuicSSH over a single path does nearly not vary when changing the size of files. Indeed if we consider only the file size larger or equal to 1024kB, this percentage only increases to 52.5%. This could probably be different if the scheduler took into account the loss rate for each path.

5.5.3 Another scheduling method: constraint-based proactive scheduling

In Multipath QUIC, a scheduler decides which path to use for sending packets. In the current version of Multipath QUIC, this is the "lowest RTT first scheduling". It assigns all the packets to the path having the lowest RTT. When the congestion window associated with this path becomes full, it then uses the next link having the lowest RTT. The previous experiments showed that this may not be optimal: when paths have significantly different performances, we may slow down all the transmission by using the second path instead of using only the fastest path. This is because using paths with different delays may cause packet reordering and thus this can lead to a form of head-of-line blocking. Indeed an application that is using Multipath QUIC expects to receive all the packets in order. But the receive buffer inside Multipath QUIC is not unlimited and thus waiting for a packet transferred to the highest-delay path may slow down the transmission.

We have implemented inside Multipath QUIC another scheduling method already existing for Multipath TCP: "Constraint-based proactive scheduling for MPTCP in wireless network" (CP scheduling) [54]. This method takes into account the difference between the RTTs of the paths and the available size in the congestion window of each path. This is shown with Multipath TCP to be better than the simple "lowest RTT first scheduling".

We use the "radical CP scheduler" approach described in this paper which consists, given we have two paths, to use first the link with the lowest RTT until congestion window becomes saturated. Then, we use the second path if the available buffer is larger than a value `B`. `B` is computed with a formula such that it is higher if the ratio $\frac{\text{RTT}_{\text{link with highest RTT}}}{\text{RTT}_{\text{link with lowest RTT}}}$ is higher and also higher if the quantity of packets sent on the first path for which no ACK already came back is high.

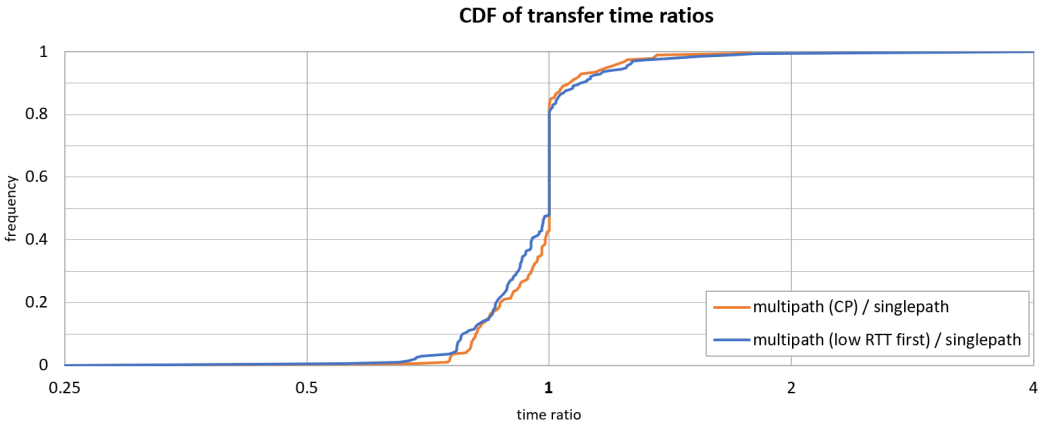

Figure 5.12: download times ratios CDF in environment without losses

Figure 5.12 presents 2 CDF: the one already shown on Figure 5.10 where we have lowest-RTT first scheduling; and another CDF that compares QuicSSH over Multipath QUIC using constrained-base proactive scheduling (CP) with QuicSSH over a single path. We can see that using CP scheduling slightly increases the proportion in which Multipath QUIC is faster or as fast than single path (83.5% instead of 81.5%). But this difference is not so significant.

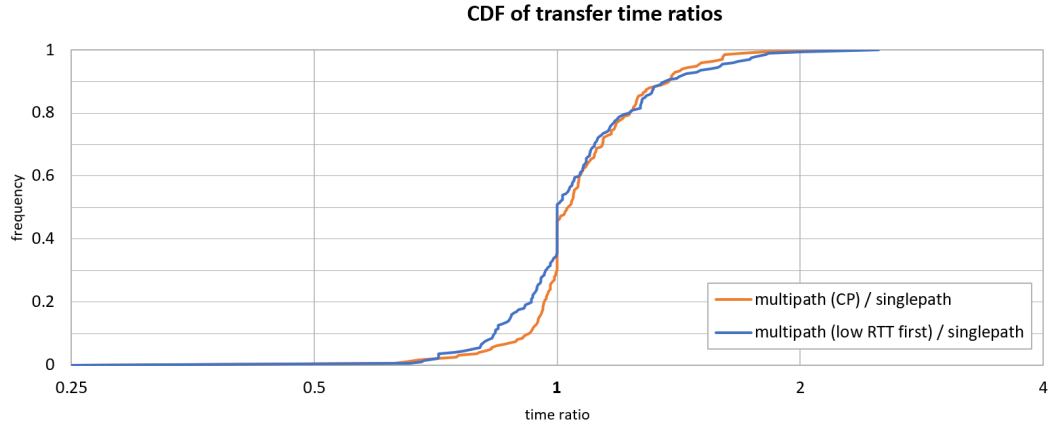

Figure 5.13: download times ratios CDF in environment with random losses up to 2%

Concerning the network conditions where random losses are allowed up to a rate of 2%, we can see on the figure 5.13 that now the performances are decreasing when using CP scheduling. We now have 45.5% of situations in which QuicSSH over multipath is faster or as fast than QuicSSH over a single path. This is below the 51% we had with lowest-RTT first scheduling.

Summary for multipath performances

fsize	no loss, low RTT first	no loss, CP	with loss, low RTT first	with loss, CP
>= 8kB	81.0%	83.5%	51.0%	45.5%
>= 128kB	85.0%	81.5%	52.5%	40.5%
>= 1024kB	87.5%	82.5%	52.5%	45.0%
= 8192kB	100%	92.5%	52.5%	42.5%

Table 5.3: Proportion in which QuicSSH over Multipath QUIC is faster than QuicSSH over a single path, depending on the size of files transferred and on the scheduling method used

Table 5.3 summarizes the performances we have with and without losses, with lowest-RTT first scheduling or with CP scheduling and for various ranges of file sizes. We clearly see that using multipath is beneficial when there is no loss and that this is particularly true when the size of the data transferred increases. Using CP scheduling does not seem a good idea as it shows less good results for larger files. Finally, when looking at the results when losses are present, we see that there is no clear winner between single path and multipath when considering the lowest-RTT first scheduling for multipath. Also having larger files does not increase performances with multipath. A possible improvement could be thus to implement a third scheduling method in multipath that compares the loss rates for all the paths and that prevents using less good paths if they are too problematic. This would be useful in order to prevent to face a form of head-of-line blocking. Then we could expect that QuicSSH over Multipath QUIC would become at least at fast as QuicSSH over a single path, and eventually faster if the links available are not too heterogeneous.

5.6 Summary

In this chapter we presented QuicSSH, a tool providing remote login and port forwarding above QUIC. Its design, inspired by SSH (security, usage, behaviour) still has several differences. Those include the possibility to have certified server key, the control messages at each new connection to the port forwarding and the usage of Telnet for the remote login.

By focusing our analysis on the port forwarding component, we show that, in most of the situations (with or without concurrency, with different network characteristics), QuicSSH is faster than TCP equivalents. We explain it by several reasons. QUIC reduces head-of-line blocking thanks to its stream multiplexing implementation. Also, QuicSSH recovers faster from losses compared to TCP based alternatives because QUIC has a more specific retransmission strategy and uses a more aggressive growth for the congestion window. Finally the implementation of QuicSSH itself has less round-trip times at connection establishment in comparison with SSH or SSF.

We also introduced Multipath QUIC, as a promising extension to the QUIC protocol, which could benefit to QuicSSH. It allows in theory faster transmissions thanks to bandwidth aggregation and it handles more quickly link failures. When there is no random loss on the network, we measure that QuicSSH using Multipath QUIC is faster or as fast than QuicSSH over a single path in the majority of the situations. But this is no more the case when we consider networks with losses. The current implementation of Multipath QUIC uses a “lowest RTT first” scheduler. We tried to implement another one that already showed good results with Multipath TCP: constrained-based proactive scheduling. But here, using this alternative scheduler results in a performance degradation. It thus needs to be improved. Some possible tracks include the development of another scheduler that takes more the loss rate into account when selecting a path. In that case, QuicSSH over Multipath QUIC should become as fast as QuicSSH over a single path (because the scheduler can make the choice to only use the best path). And it could eventually become faster than QuicSSH over a single path if the paths available are not too heterogeneous by taking advantages of both links.

Chapter 6

Application 2: QuicVPN

In this chapter, we present our second application: QuicVPN. Similarly as for QuicSSH, it takes place in multiple parts. We start by describing our program and its architecture. Then, we review its performance, by comparing it to alternative solutions over TCP or UDP. Finally, as with the Multipath extension of QuicSSH, we also give some possible extensions to introduce new features in our tool.

6.1 Program description and architecture

QuicVPN is a tool to build virtual private networks running on top of QUIC. It uses the same technology stack as QuicSSH.

In terms of functionality, as any other VPN, it allows the exchange of data between two private networks using QUIC to secure the transmission on public network (Figure 6.1). It works in an authenticated client-server mode. After checking the identity of the other side, both endpoints listen to the TCP/UDP traffic coming from the private network. They encapsulate data into QUIC packets and transmit them over the public network. Each endpoint will, as its turn, read the traffic from the QUIC packets and inject the TCP/UDP payload into their internal network.

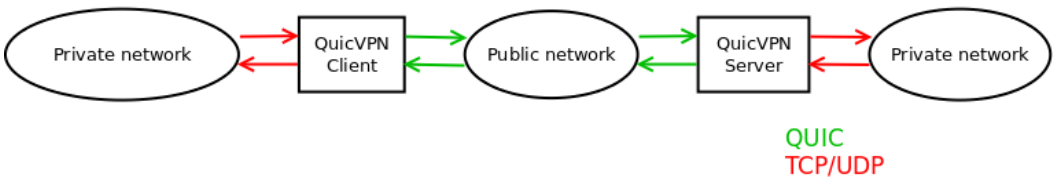

Figure 6.1: High level overview of the VPN principle

Its architecture is modular and structured around a main routine, which starts the execution of four concurrent modules (Figure 6.2).

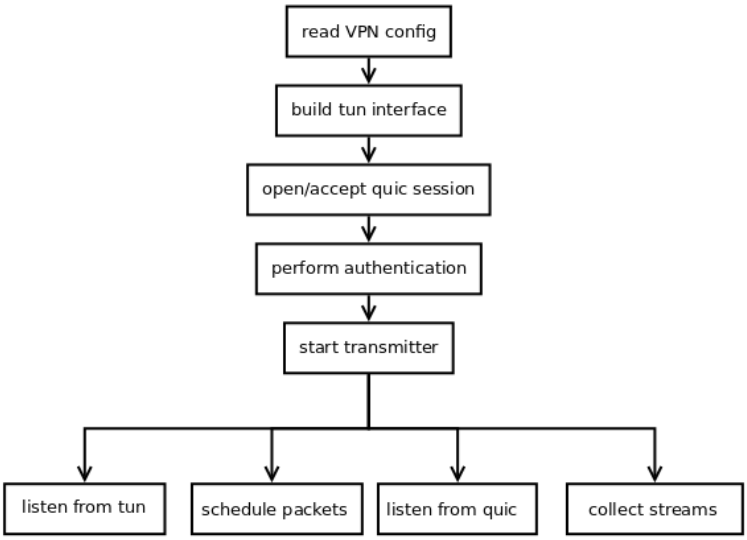

Figure 6.2: Main routine of our vpn

When we start our VPN, the first operation is to read the configuration file. It contains information such as the IP and port to establish the VPN, if we would like to build a client or a server, the public certificates of the endpoints we accept to connect and some network configurations.

Then, just after that, we build and configure a tun interface. Tun interfaces are a virtual driver exposed by the Linux kernel to allow sending and receiving network traffic from a user space program (here, our VPN). Every packet sent by the network to the "tun" interface is read by our program. Every packet sent by our program to the "tun" interface is relayed to the internal network.

Depending on whether we are a client or server, we then open or accept a QUIC session.

After that, we perform authentication with our applicative protocol defined in Chapter 4. To allow a connection, we compare the certificate received to a list of allowed client/server certificates defined in the configuration. Then, we continue execution or reject the connection if it is not in the list.

Finally, we can start tunnelling data by launching our modules.

- One listens IP traffic coming from the internal network and put it in a queue.
- Another schedules the packets that needs to be sent, encapsulate them and write to a QUIC stream.
- A third one listens from QUIC traffic, decapsulates it and sends the IP payload to the internal network.
- Finally, we also collect inactive streams.

6.1.1 Listening from the network

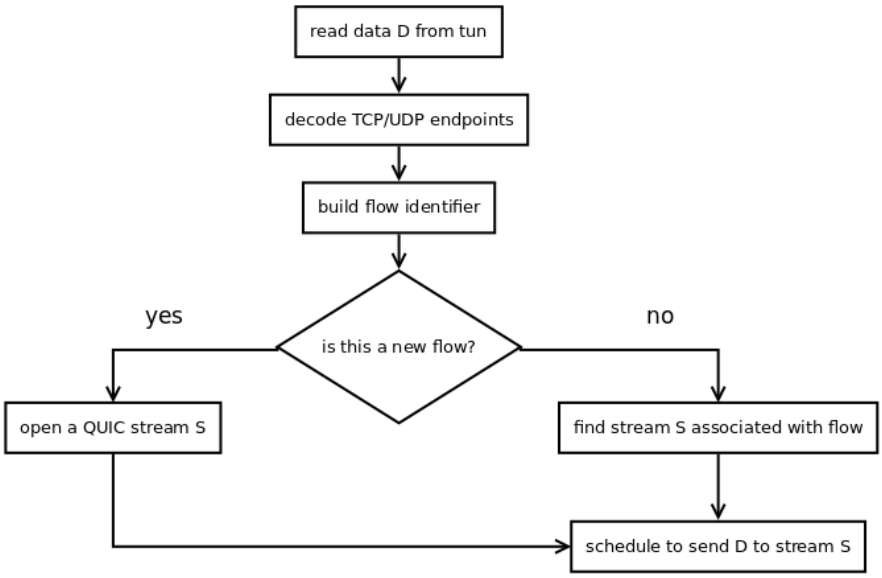

Figure 6.3: Listen from tun

To listen from the internal network, we simply read IP datagrams from the tun interface. Then, for every packet we receive, we decode it to identify the endpoints and use a hash method to build a unique flow identifier to ensure than every TCP/UDP session will go through a different QUIC stream. After that, we check if we already mapped this flow identifier to a stream. If not, we ask QUIC to open a new one. Then, every data that we read on the tun interface coming from this flow will be scheduled to be sent on the same stream.

6.1.2 Schedule sending packets

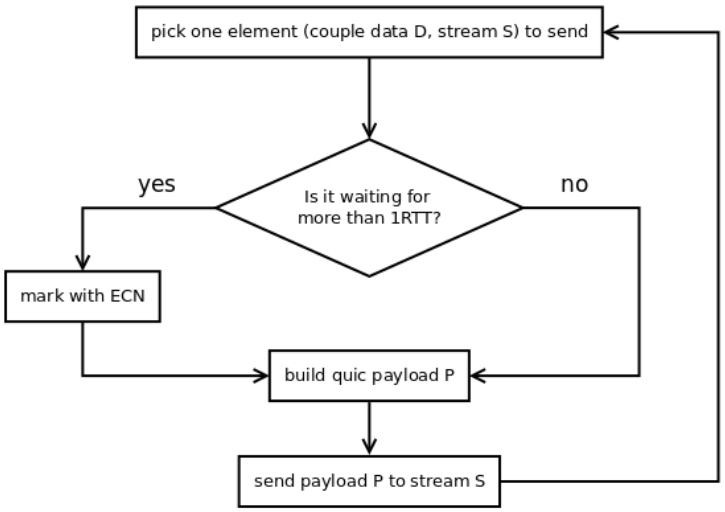

Figure 6.4: Scheduler

Then comes the scheduler, which is a crucial part of our architecture. Its goal is to get the information to send to the network (IP datagram, stream to use), then send it.

It works in three parts.

1. Packet selection

First, there is the selection step. By default, we pick from every datagrams to send as they come, to preserve the initial priority of packets. But this is easily extensible and can support other scheduling policies.

2. Meltdown avoidance

As QuicVPN also use a reliable transmission medium to encapsulate packets (QUIC), it should be impacted by the same problems as TCP based tunnels (highlighted in Section 2.3.2).

But as we don't want to apply any restriction to the usage of our VPN, or need some workaround or particular shaping on the network to make it adapt bandwidth properly, such as it's the case with TCP, we implemented a new system to avoid the meltdown.

By flagging with ECN all incoming packets waiting in the queue for more than 1RTT, we indicate the hosts to reduce their sending pace. This allows them to adapt on the available bandwidth and never send faster than the rate of the VPN endpoint.

We selected 1 RTT for the threshold as, in the worst case, transported traffic is implemented without pacing, while QUIC has this feature. In that situation, hosts will send bursts where the last packet will have to wait 1 RTT. That is why we cannot set it lower. Smaller values will drop packets too early and reduce bandwidth. Larger ones will allow the hosts having too big congestion windows and create some queues. If we were sure that the packets arriving in QuicVPN were properly paced, we could put thus threshold much lower. If pacing rates match, any packet would be sent immediately when it comes to the tunnel. But, we don't want to make this assumption on the kind of transported traffic.

Despite it should still theoretically be impacted by the second kind of meltdown, as it uses similar retransmission time computation as TCP [55], we hope that QUIC transmission strategies, that support larger selective acknowledgement ranges will completely avoid this problem to happen.

3. Building payload

After selecting and marking incoming datagrams, we need to put them in a QUIC packet.

The complexity of this step is to transmit IP datagrams over a QUIC stream. As QUIC is a byte stream medium, it does not allow to set a boundary in the stream of data read. We solved this by prefixing each IP datagram by its length (on 32bits, big endian) before writing to QUIC stream. This helps the receiver to distinguish subsequent datagrams. To optimise performance, we just have to be sure than an IP datagram is able to fit in a QUIC single packet. That is why our configuration defines a default MTU of 1150B, tuned according to the fixed size of QUIC datagrams in the `quic-go` implementation.

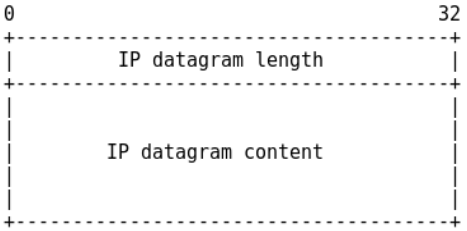

Figure 6.5: Encoding of an IP datagram in a QUIC payload

After this step, we can send the packets to the network on the right QUIC stream.

6.1.3 Read from the network

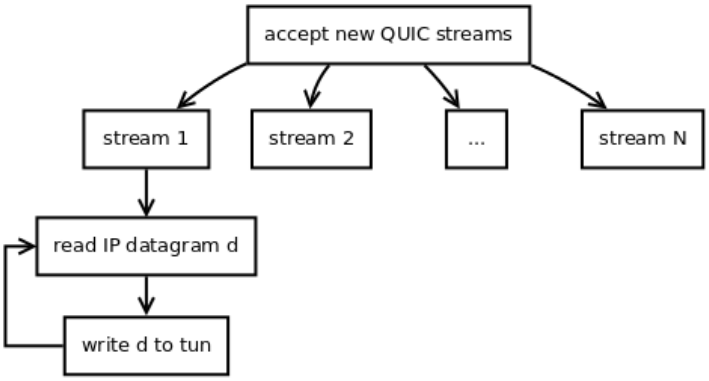

Figure 6.6: Listen from QUIC

Then, the opposite path, from the QUIC stream to the internal network is straightforward. Whenever we accept a new QUIC stream, we read data on it, extract the IP datagram and write it to the tun interface to transmit it to the internal network.

6.1.4 Collect unused streams

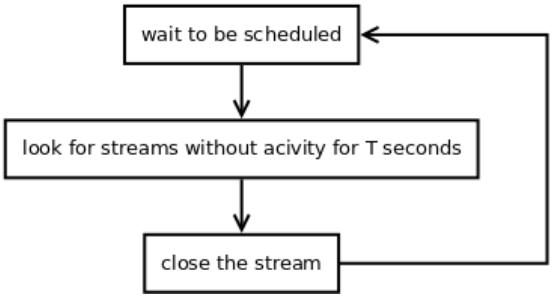

Figure 6.7: Collect unused streams

As streams consume resources, we don't keep them for closed or inactive TCP/UDP. We implemented a system that automatically closes streams inactive for more than 1 second. We don't need to look at the FIN or RST flags of TCP sessions, because QUIC streams can be reopened on demand, without any packet loss if we close a stream too early.

6.2 Performance

Now that we have a working system designed to avoid some parts of the meltdown, we can start to measure its performance. We proceed in two parts. First, we review general efficiency of our ECN system to avoid meltdown. Then, we check general performance in different real-world use cases, by comparing it with common implementations of both TCP, but also UDP tunnels, using OpenVPN[1] (which can support both modes).

[1] `https://openvpn.net/`

6.2.1 Meltdown

To ensure than our system correctly reduces the meltdown, we measured the bandwidth and round-trip time of a 50sec TCP iperf download inside a QuicVPN tunnel. In our test setup (Figure 3.1), we shaped the link to have 10Mbps of capacity, 100ms of RTT and no losses. We compared the results in two situations.

Without ECN

If we use a QuicVPN variant with the ECN system disabled (Figure 6.8), we can see that QuicVPN is subject to meltdown. iperf TCP packets are arriving faster and faster. They don't experience any loss. The congestion window (measured with iperf3) and RTT (measured with Wireshark) grows indefinitely, until timers expire (around 1second, due to the minimum value of RTO [31]). This means that the inner sessions don't adapt on available bandwidth.

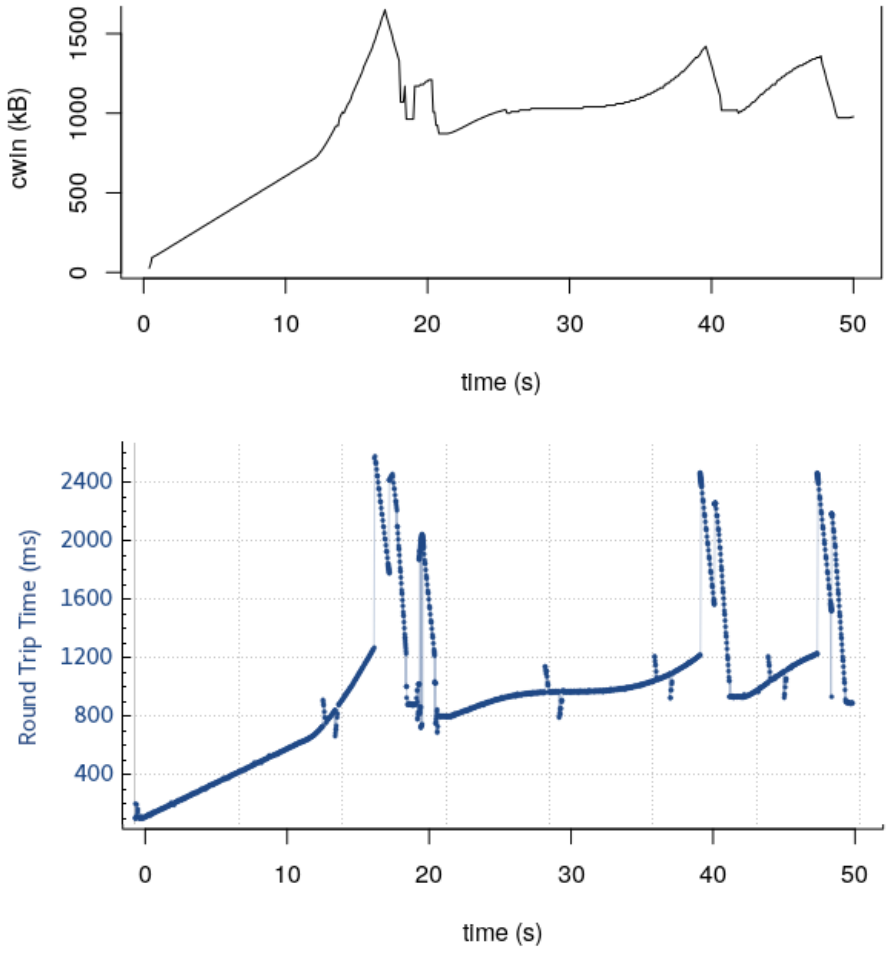

Figure 6.8: CWIN & RTT of iperf3 download over QuicVPN without ECN

With ECN system

If we repeat the experiment, but this time, with our ECN system enabled (Figure 6.9), results are better. When packets start to accumulate, congestion window does not have the time to grow. When packets wait too long in the sending queue, they are marked with ECN.

Without having any particular shaping, we reduced the RTT to at most twice the RTT of the link. We thus only induce a single RTT of overhead, coming from our ECN system, as it needs to wait before flagging the packets. And, it rarely exceeds this value.

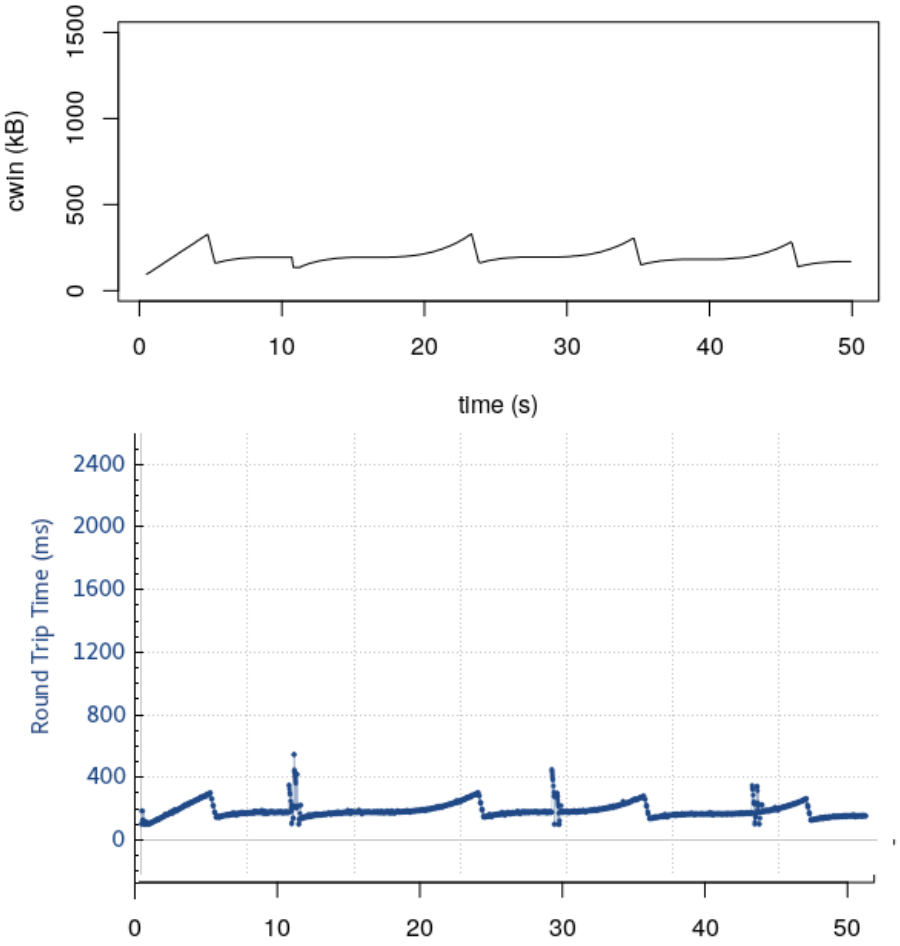

Figure 6.9: CWIN & RTT of iperf3 download over QuicVPN with ECN

Moreover, we can see that our ECN system allows the iperf download to get most of the speed of the bottleneck link (Figure 6.10). It is thus more efficient that shaping based solutions that limit the bandwidth of the network.

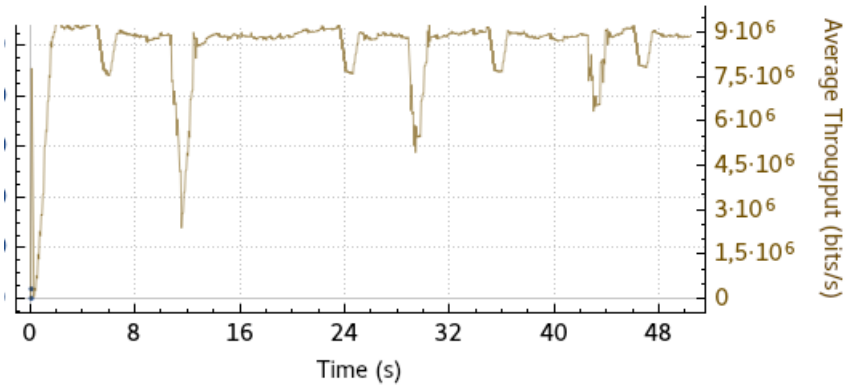

Figure 6.10: Bandwidth of iperf3 download over QuicVPN with ECN

6.2.2 Use case 1: Downloading small files

After reviewing the capability of our system to handle meltdown, we can start investigating the performance of our tunnel when running different applications inside it.

Using our experimental framework and the reviewed parameter grid, we simulate 100 iterations of different use cases. The first one is the download with apache benchmark of batches that contains 16 same sized small files (from 8 to 1024kB).

We compared the performance of our QuicVPN against different alternatives: OpenVPN in TCP mode[2] or UDP mode. We tested different QuicVPN settings, by allowing, or not concurrent downloads.

We reported the time needed to download all the files and divided it by the time needed for a direct transfer, without any tunnel and reported the ratio in a CDF. A value smaller than 1 indicates a tunnelling technology is faster than a direct transfer.

When we perform only sequential downloads (Figure 6.11), the fastest solution is QuicVPN. It overcomes the performance of OpenVPN (in both modes). It is even faster than a direct transfer. This is linked to the aggressiveness of QUIC, which is able to react to losses twice faster than a single TCP sessions.

Moreover, there is only a small negative impact when activating our ECN system (denoted as QuicVPN + ECN). It only reduces bandwidth when we start sending too fast.

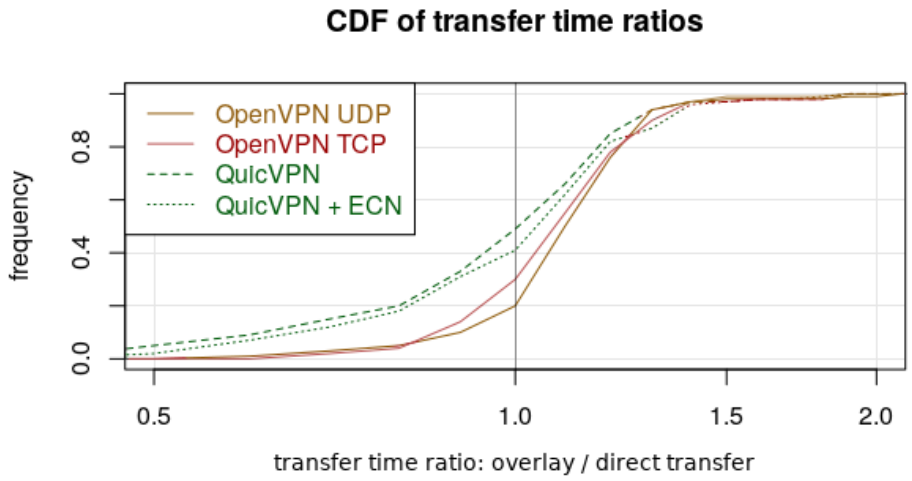

Figure 6.11: Download of 16 small files: ratio over direct transfer

Adding concurrency (Figure 6.12) degrades the performance of QuicVPN, making it far less efficient than UDP or direct transfer (ratio bigger than 1). This is a consequence of the QUIC poor performance when using multiple streams in a lossy environment (as already identified in Section 3.3.3).

But it is still better than solutions using a single stream (OpenVPN TCP and QuicVPN limited to one stream). As for QuicSSH, we explain this by the reduction of Head-of-line blocking. Bundling multiple sessions inside a single TCP or QUIC stream is not good for the performance of a VPN, as any loss would block the entire connection. Moreover, its aggressive congestion window growth due to MulTCP also makes QUIC always faster than TCP, even when comparing single-streamed implementation together.

[2]With the Naggle algorithm deactivated, as advised by the developers

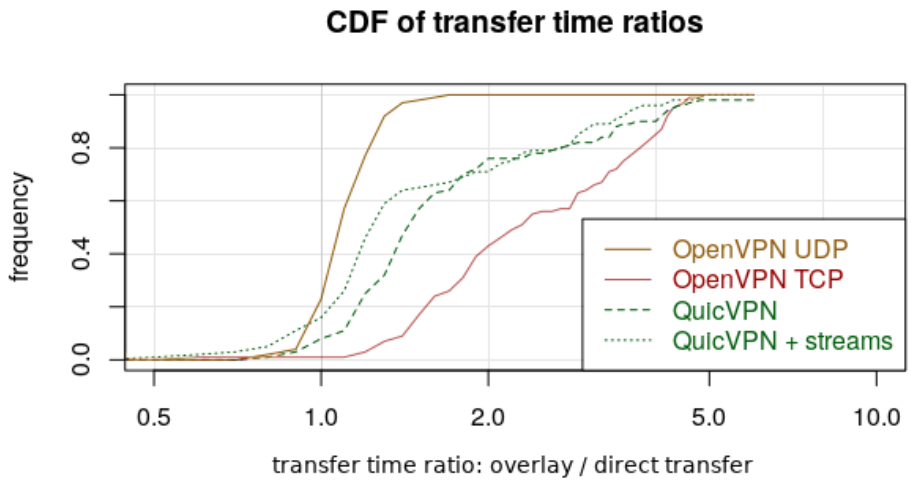

Figure 6.12: Download of 16 small files: ratio over direct transfers (with 4 concurrent downloads)

6.2.3 Use case 2: Large downloads

We repeated the experiments, but with larger downloads by using only batches of 2MB files (Figure 6.13).

This shows the same trends as with the small files. With sequential downloads, QuicVPN is the fastest solution and adding ECN does not lower too much the performance.

With concurrent downloads, QuicVPN with a single stream is better than OpenVPN TCP and less good than QuicVPN with multiple streams or OpenVPN in UDP mode, for the same reasons.

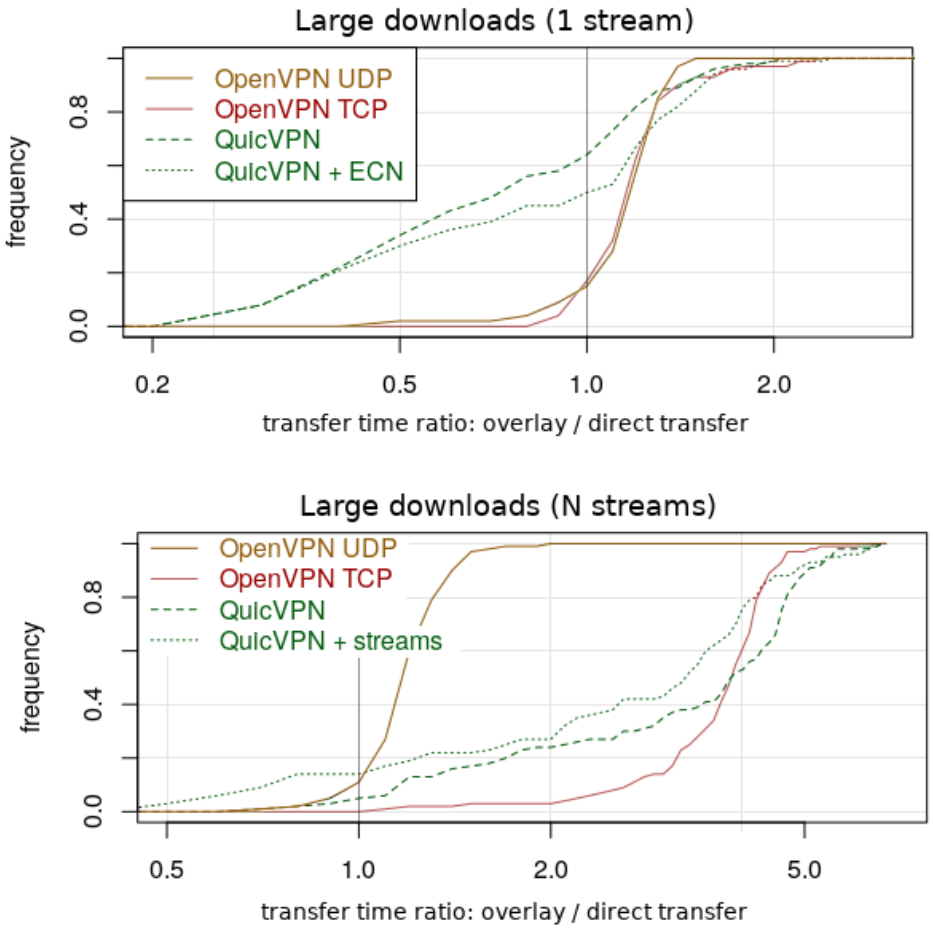

Figure 6.13: Large files

6.2.4 Use case 3: Interactive applications

Even if bandwidth is an important element, one of the major reasons behind the existence of QUIC is the need to improve latency. That is why we also need to study the effective delay perceived by applications when running inside a QUIC based VPN.

Using the same framework and leveraging the D-ITG tool, we ran 100 iterations of different interactive applications (VoIP G.711.2, Counter strike[56] in active mode, Telnet and a Quake simulation[57]) running concurrently to a 10sec iperf download (Figure 6.14).

Here, we only compare OpenVPN in TCP or UDP mode and QuicVPN limited to one stream or using multiple streams, considering our ECN module as always on.

QuicVPN vs direct transfer

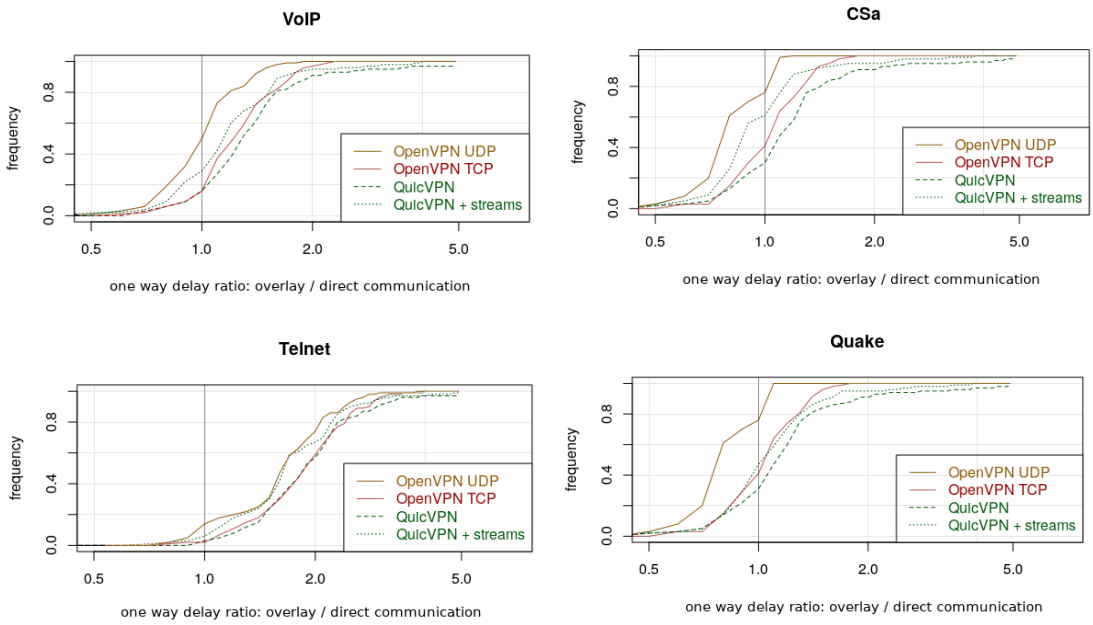

Figure 6.14: Ratio of the application one-way delay using an overlay over a direct connection

From our results (Figure 6.14), we can observe several important elements.

OpenVPN in UDP mode is always the fastest solution. For UDP based flows, it is very close to the reactivity of a direct transfer (ratio close than 1). For Telnet, it varies a bit, due to congestion control, but is 50% of the times slower and 50% faster, due to the randomness of the delay when subject to uniform losses.

We remark a big advantage of using one QUIC stream per flow. This brings the performance of QuicVPN closer to OpenVPN in UDP than in TCP mode. In that case, QuicVPN with streams achieves an average round-trip time not more than twice larger than a direct transfer.

QuicVPN vs OpenVPN

If we compare OpenVPN experiments with TCP or UDP transfer instead of comparing with a direct communication, we can observe than:

- Versus TCP (Figure 6.15), QuicVPN is faster in nearly all experiments, probably due to the reduction of meltdown and head-of-line blocking
- Versus UDP (Figure 6.16), QuicVPN always slower for counter-strike and Quake traffic, that are, by nature, unreliable. Thus, the retransmission layer of QUIC only induces an overhead. But it's not the case for voIP and Telnet, where QUIC retransmissions help to achieve lower delay.

In both cases, QuicVPN is not more than twice slower than OpenVPN in TCP or UDP mode, as it was never more than twice slower than a direct transfer.

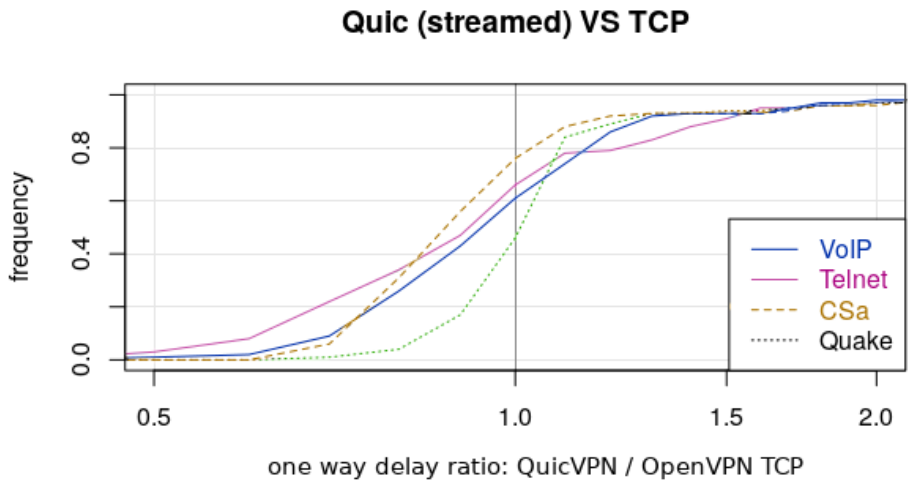

Figure 6.15: Streamed QuicVPN vs TCP

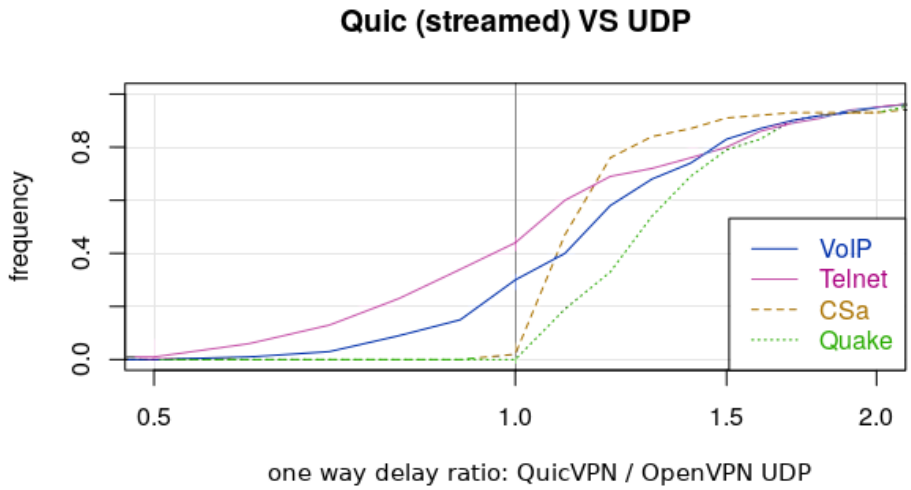

Figure 6.16: Streamed QuicVPN vs UDP

6.3 Going further

One big advantage of VPNs compared to other transport overlays is that they are able to transparently carry one kind of traffic over another. Consequently, any improvement made to the tunnelling technology directly impacts the transported payload.

Thanks to the user space nature of QUIC, our VPN is extensible, capable of bringing further improvement to the transported packets.

6.3.1 Prioritisation & Dynamic QOS

One easy way to improve our results is to extend our scheduler. Currently, it is implemented using a FIFO queue. But, we can also use many other policies. We could, for instance, decide to prioritise VoIP over Telnet or to give more bandwidth to any other flow at the VPN level. We

could even decide to adapt it at runtime and signal priorities to the other endpoint. This can help to achieve much more dynamic settings depending on the current state of the network.

As a proof of concept, we have extended our code with the possibility to prioritise based on port (source for clients, destination for servers). Each time we get a packet, we sort the sending queue (using a stable strategy, to avoid reordering) then, pop and send one packet. This gives more bandwidth for the streams with a higher priority. As packets in streams in the lowest priority will be popped later, they will have to wait. If this waiting time exceeds one RTT, they are marked by ECN and lower their sending rate.

If we run 5 concurrent downloads using iperf (for 30 seconds on 10Mbps network with 50ms of RTT no losses inside a QuicVPN with stream prioritisation), we can see that bandwidth effectively splits across the streams depending on their priority. Stream 1 gets most of the bandwidth, stream 2 a bit less and all the other streams, nearly nothing (Figure 6.17).

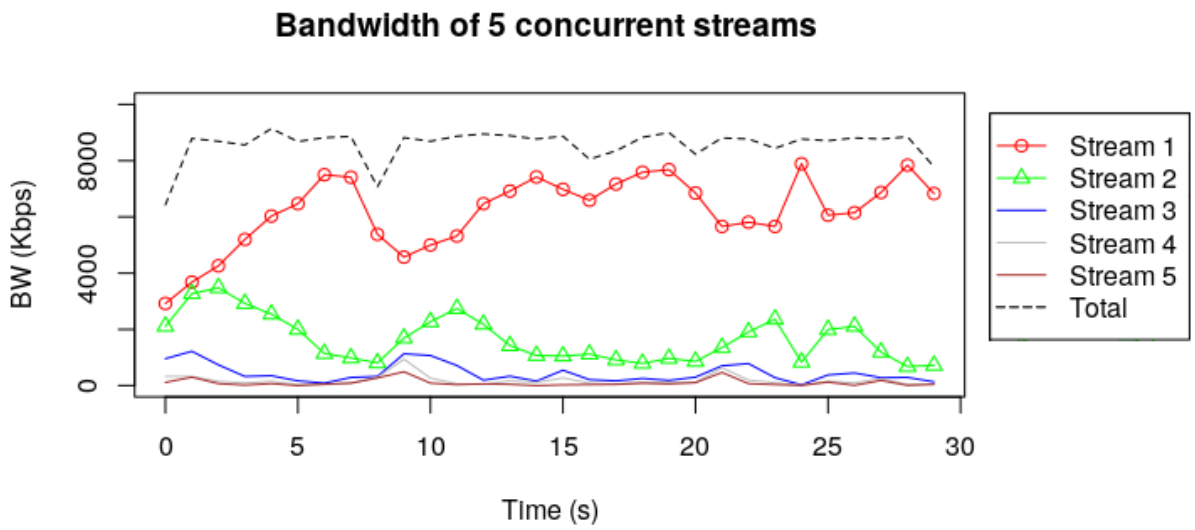

Figure 6.17: Illustration of the impact on the TCP streams bandwidth

6.3.2 Multipath

As with QuicSSH, we can also use solutions such as Multipath QUIC to have the possibility to benefit from multiple links. It can help to provide higher throughput, better delay and faster handover, without changing anything in the end hosts.

6.3.3 Unreliable streams

Ongoing research on the usage of unreliable streams in QUIC [58] can also improve our QuicVPN implementation.

Having pure unreliable QUIC based VPN would bring in the possibility to build VPNs that have a performance comparable to a direct transfer. They would not be subject to the meltdown and can still have some new features: multipath, error correcting codes or timing deadlines. We could even imagine that those features can apply on specific streams. This can be based on an inspection of traffic running inside the tunnel or the state of the network.

6.4 Summary

This chapter presented QuicVPN, a tool to make virtual private networks using QUIC to relay packets.

Based on an in-depth analysis in different setups, we have shown that it is faster and more reactive than TCP tunnels in most of the case, either with or without concurrency. This is a combination of three main reasons. By using multiple streams, QuicVPN copes better with head-of-line blocking, which impacts the reactivity of single stream TCP based tunnels. It is also faster, as the congestion window growth is regulated by the MulTCP algorithm, which is here twice more aggressive than that of a single TCP session. Finally, its ECN-based meltdown avoidance system also play a role to reduce the delay in networks where the bottleneck is on the WAN link.

We have also presented several promising extensions to make it even faster. As QuicSSH, QuicVPN can benefit from multipath capabilities. It can also implement applicative level dynamic scheduling or even decide to use unrealiable streams.

But it is still far from a perfect solution. Despite it overcomes TCP tunnels in many case, they don't constitute the most widespread implementation of VPNs. UDP tunnels are much more deployed and overcome the performance of QuicVPN in many aspects. They are faster, more reactive and don't need system to avoid an hypothetical meltdown.

QuicVPN can thus be considered as a good alternative for TCP based tunnels, but only where UDP tunnels are not applicable. Examples includes the need for multipath solutions. As no good implementation of this system currently exist on top of UDP, we could imagine to use QuicVPN in this situation.

Chapter 7

Conclusion

We conclude this thesis by summarising the lessons we have learned by building and measuring our two QUIC based programs. We discovered that QUIC can be a good solution to develop new applications. Its simplicity, due to its user-space nature, and its efficient design led us to quickly build software that are better than their TCP equivalents.

The first software implemented on QUIC is QuicSSH. Its port forwarding is more efficient in nearly all situations than the TCP equivalents, with and without concurrency. We explain this by three major reasons. First, QUIC reduces head-of-line blocking thanks to stream multiplexing, while SSH bundles many connections in a single TCP session. Moreover, QUIC is faster in comparison to TCP to recover from losses. Finally, it also helps to download small files using fewer round-trips thanks to an optimised connection establishment.

QuicVPN is our second software that we have designed and implemented. For the same reasons as QuicSSH, using a multi-streamed QUIC VPN is always faster than a VPN over TCP such as OpenVPN, which uses a single session to carry its data.

Both software are even able to support new functionalities, which would be very hard to develop in other transport protocols. We included in QuicSSH the possibility to work on multiple paths. We allowed QuicVPN to notify the inner packets of the network congestion. And, we also experimented some kinds of dynamic scheduling.

But, the QUIC protocol is still in its early days. We discovered that is suffers from several flaws. In our QUIC implementation, we remarked the impossibility to cope well with jitter, the lack of support for some important functions, such as the session resumption or the path MTU discovery. We also identified that, despite stream multiplexing is promising, it is not a perfect replacement for the bundle of multiple TCP sessions, largely used nowadays. The MulTCP system implemented in `quic-go` to simulate the combined congestion control of multiple flows is not perfect. It is not fair and does not work when concurrency is pushed at a high level. This is why QuicVPN tunnelling and QuicSSH port forwarding does not compete with direct transfers.

That is the same for the multipath extension. Despite it looks really promising, we remarked some flows in its scheduler. The current default "lowest RTT first" strategy does not work when paths are too heterogeneous. And, its improvement to "constrained-based proactive scheduling", which is the state-of-the-art solution for Multipath TCP, does not bring more benefits. It needs some more reflection to take into account the amount of losses, and to not send data on alternative paths, if they are too lossy.

We also found that, despite QUIC is gaining more and more traction, it should still be considered with care. It is not a perfect replacement for every transport protocol. We've shown that comparing our QuicVPN solution with OpenVPN in UDP mode lead to generally better result for the UDP based solution. All applications made on UDP may thus sometimes not benefit from an implementation over QUIC.

Bibliography

[1] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol–http/1.1," tech. rep., 1999.

[2] G. Maier, A. Feldmann, V. Paxson, and M. Allman, "On dominant characteristics of residential broadband internet traffic," in *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, pp. 90–102, ACM, 2009.

[3] J. Postel, "Transmission control protocol," STD 7, RFC Editor, September 1981. `http://www.rfc-editor.org/rfc/rfc793.txt`.

[4] U. Goel, M. Steiner, M. P. Wittie, M. Flack, and S. Ludin, "Http/2 performance in cellular networks," in *ACM MobiCom*, 2016.

[5] J. Roskind, "Quic (quick udp internet connections): Multiplexed stream transport over udp," *Technical report, Google*, 2013.

[6] J. Iyengar and M. Thomson, "Quic: A udp-based multiplexed and secure transport," Internet-Draft draft-ietf-quic-transport-09, IETF Secretariat, January 2018. `http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-09.txt`.

[7] M. Seemann and L. Clemente, "quic-go." `https://github.com/lucas-clemente/quic-go`. Online; accessed 28 February 2018.

[8] E. Rescorla, "The transport layer security (tls) protocol version 1.3," Internet-Draft draft-ietf-tls-tls13-23, IETF Secretariat, January 2018. `http://www.ietf.org/internet-drafts/draft-ietf-tls-tls13-23.txt`.

[9] W.-T. C. Adam Langley, "Quic crypto." `https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit`. Online; accessed 31st May 2018.

[10] S. Ohtsu, "Technical overview of quic." `https://fr.slideshare.net/shigeki_ohtsu/quic-overview`. Online; accessed 19 February 2018.

[11] P. Karn and C. Partridge, "Improving round-trip time estimates in reliable transport protocols," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 4, pp. 364–373, 1991.

[12] J. Iyengar and I. Swett, "Quic loss recovery and congestion control," Internet-Draft draft-tsvwg-quic-loss-recovery-01, IETF Secretariat, December 2015. `http://www.ietf.org/internet-drafts/draft-tsvwg-quic-loss-recovery-01.txt`.

[13] S. Ha, I. Rhee, and L. Xu, "Cubic: a new tcp-friendly high-speed tcp variant," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 64–74, 2008.

[14] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, *et al.*, "Bbr: congestion-based congestion control," *Communications of the ACM*, vol. 60, no. 2, pp. 58–66, 2017.

[15] J. Gettys and K. Nichols, "Bufferbloat: Dark buffers in the internet," *Queue*, vol. 9, no. 11, p. 40, 2011.

[16] M. Hock, R. Bless, and M. Zitterbart, "Experimental evaluation of bbr congestion control," in *Network Protocols (ICNP), 2017 IEEE 25th International Conference on*, pp. 1–10, IEEE, 2017.

[17] Y. Cui, T. Li, C. Liu, X. Wang, and M. Kühlewind, "Innovating transport with quic: Design approaches and research challenges," *IEEE Internet Computing*, vol. 21, no. 2, pp. 72–76, 2017.

[18] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, *et al.*, "The quic transport protocol: Design and internet-scale deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pp. 183–196, ACM, 2017.

[19] A. Srivastava, *Performance Analysis of QUIC Protocol under Network Congestion*. PhD thesis, WORCESTER POLYTECHNIC INSTITUTE, 2017.

[20] T. Ylonen and C. Lonvick, "The secure shell (ssh) protocol architecture," RFC 4251, RFC Editor, January 2006. `http://www.rfc-editor.org/rfc/rfc4251.txt`.

[21] T. Ylonen and C. Lonvick, "The secure shell (ssh) transport layer protocol," RFC 4253, RFC Editor, January 2006. `http://www.rfc-editor.org/rfc/rfc4253.txt`.

[22] R. Canetti and H. Krawczyk, "Analysis of key-exchange protocols and their use for building secure channels," in *International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 453–474, Springer, 2001.

[23] T. Ylonen and C. Lonvick, "The secure shell (ssh) authentication protocol," RFC 4252, RFC Editor, January 2006. `http://www.rfc-editor.org/rfc/rfc4252.txt`.

[24] T. Ylonen and C. Lonvick, "The secure shell (ssh) connection protocol," RFC 4254, RFC Editor, January 2006. `http://www.rfc-editor.org/rfc/rfc4254.txt`.

[25] E. Rescorla and N. Modadugu, "Datagram transport layer security version 1.2," RFC 6347, RFC Editor, January 2012. `http://www.rfc-editor.org/rfc/rfc6347.txt`.

[26] B. P. Lee, R. K. Balan, L. Jacob, W. K. G. Seah, and A. L. Ananda, "Tcp tunnels: avoiding congestion collapse," in *Local Computer Networks, 2000. LCN 2000. Proceedings. 25th Annual IEEE Conference on*, pp. 408–417, IEEE, 2000.

[27] O. Titz, "Why tcp over tcp is a bad idea." `http://sites.inka.de/bigred/devel/tcp-tcp.html`. Online; accessed 17 February 2018.

[28] "Openvpn." `https://en.wikipedia.org/wiki/OpenVPN`. Online; accessed 18 May 2018.

[29] L. S. Brakmo and L. L. Peterson, "Tcp vegas: End to end congestion avoidance on a global internet," *IEEE Journal on selected Areas in communications*, vol. 13, no. 8, pp. 1465–1480, 1995.

[30] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "Bbr: Congestion-based congestion control," *Queue*, vol. 14, no. 5, p. 50, 2016.

[31] V. Paxson, M. Allman, J. Chu, and M. Sargent, "Computing tcp's retransmission timer," RFC 6298, RFC Editor, June 2011. `http://www.rfc-editor.org/rfc/rfc6298.txt`.

[32] O. Honda, H. Ohsaki, M. Imase, M. Ishizuka, and J. Murayama, "Understanding tcp over tcp: effects of tcp tunneling on end-to-end throughput and latency," in *Performance, Quality of Service, and Control of Next-Generation Communication and Sensor Networks III*, vol. 6011, p. 60110H, International Society for Optics and Photonics, 2005.

[33] C. Paasch, R. Khalili, and O. Bonaventure, "On the benefits of applying experimental design to improve multipath tcp," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pp. 393–398, ACM, 2013.

[34] Y.-C. Chen, Y.-s. Lim, R. J. Gibbens, E. M. Nahum, R. Khalili, and D. Towsley, "A measurement-based study of multipath tcp performance over wireless networks," in *Proceedings of the 2013 conference on Internet measurement conference*, pp. 455–468, ACM, 2013.

[35] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, p. 19, ACM, 2010.

[36] A. Werner, "Linux traffic control-implementation overview," tech. rep., 1998.

[37] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, and K. Prabhu, "iperf3: A tcp, udp, and sctp network bandwidth measurement tool." `https://github.com/esnet/iperf/`.

[38] The Apache Software Foundation, "ab - apache http server benchmarking tool." `https://httpd.apache.org/docs/2.4/en/programs/ab.html`.

[39] S. Avallone, S. Guadagno, D. Emma, A. Pescape, and G. Ventre, "D-itg distributed internet traffic generator," in *Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings. First International Conference on the*, pp. 316–317, IEEE, 2004.

[40] G. Harris, D. Ovsienko, M. Richardson, and R. Francoise, "the tcpdump network dissector." `https://github.com/the-tcpdump-group/tcpdump`.

[41] J. C. Helton and F. J. Davis, "Latin hypercube sampling and the propagation of uncertainty in analyses of complex systems," *Reliability Engineering & System Safety*, vol. 81, no. 1, pp. 23–69, 2003.

[42] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, "The macroscopic behavior of the tcp congestion avoidance algorithm," *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 3, pp. 67–82, 1997.

[43] L. Eggert, J. Heidemann, and J. Touch, "Effects of ensemble-tcp," *ACM SIGCOMM Computer Communication Review*, vol. 30, no. 1, pp. 15–29, 2000.

[44] J. Crowcroft and P. Oechslin, "Differentiated end-to-end internet services using a weighted proportional fair sharing tcp," *ACM SIGCOMM Computer Communication Review*, vol. 28, no. 3, pp. 53–69, 1998.

[45] P. Mulroy, S. Appleby, M. Nilsson, and B. Crabtree, "The use of multcp for the delivery of equitable quality video," in *Packet Video Workshop, 2009. PV 2009. 17th International*, pp. 1–9, IEEE, 2009.

[46] F.-C. Kuo and X. Fu, "Probe-aided multcp: an aggregate congestion control mechanism," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 1, pp. 17–28, 2008.

[47] "Secure Socket Funneling - Network tool and toolkit." `https://securesocketfunneling.github.io/ssf/`. Online; accessed 19 February 2018.

[48] Q. De Coninck and O. Bonaventure, "Multipath quic: Design and evaluation," in *13th International Conference on emerging Networking EXperiments and Technologies (CoNEXT 2017). https://multipath-quic.org*, 2017.

[49] SSH Communications Security, "Telnet and ssh as a secure alternative." https://www.ssh.com/ssh/telnet. Online; accessed 28 February 2018.

[50] M. Thomson and S. Turner, "Draft: Using transport layer security (tls) to secure quic." https://quicwg.github.io/base-drafts/draft-ietf-quic-tls.html. Online; accessed 28 April 2018.

[51] J. Iyengar and I. Swett, "Quic loss detection and congestion control," Internet-Draft draft-ietf-quic-recovery-11, IETF Secretariat, April 2018. http://www.ietf.org/internet-drafts/draft-ietf-quic-recovery-11.txt.

[52] T. Everts, "The performance beacon: The web performance, analytics, and optimization blog." https://www.soasta.com/blog/page-bloat-average-web-page-2-mb/. Online; accessed 7 May 2018.

[53] Q. D. Coninck, "mp-quic." https://github.com/qdeconinck/mp-quic. Online; accessed 30 April 2018.

[54] B.-H. Oh and J. Lee, "Constraint-based proactive scheduling for mptcp in wireless networks," *Computer Networks*, vol. 91, pp. 548–563, 2015.

[55] J. Iyengar and I. Swett, "Quic loss detection and congestion control." https://quicwg.github.io/base-drafts/draft-ietf-quic-recovery.html#rfc.section.3.3.3. Online; accesst 1 June 2018.

[56] S. Avallone, A. Pescape, and G. Ventre, "Distributed internet traffic generator (d-itg): analysis and experimentation over heterogeneous networks," in *Poster at International Conference on Network Protocols, ICNP*, 2003.

[57] D. Emma, A. Pescape, and G. Ventre, "Analysis and experimentation of an open distributed platform for synthetic traffic generation," in *Distributed Computing Systems, 2004. FTDCS 2004. Proceedings. 10th IEEE International Workshop on Future Trends of*, pp. 277–283, IEEE, 2004.

[58] P. Tiesel, M. Palmer, B. Chandrasekaran, A. Feldmann, and J. Ott, "Considerations for unreliable streams in quic," Internet-Draft draft-tiesel-quic-unreliable-streams-00, IETF Secretariat, September 2017. http://www.ietf.org/internet-drafts/draft-tiesel-quic-unreliable-streams-00.txt.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**