Department of Computer Science and Engineering
Islamic University of Technology (IUT)

Seamless Service Migration in Cloud Edge Networks with QUIC

By
Zibran Zarif Amio - 180041209
Fida Waseque Choudhury - 180041215
Mohammed Mohaimen - 180041217

Supervised by
Dr. Muhammad Mahbub Alam
Professor
Computer Science and Engineering
Islamic University of Technology

Co-supervised by
S.M. Sabit Bananee
Lecturer
Computer Science and Engineering
Islamic University of Technology

Declaration of Authorship

This is to certify that the research presented in this thesis is the result of analysis completed by **Zibran Zarif Amio**, **Fida Waseque Choudhury** and **Mohammed Mohaimen** under the supervision of **Muhammad Mahbub Alam**, Professor, Department of Computer Science and Engineering (CSE), Islamic University of Technology (IUT), Dhaka, Bangladesh. Additionally, it is stated that neither this thesis nor any portion of this thesis has ever been presented anywhere for a degree or diploma. A list of references is provided, and the text includes acknowledgements for information that was taken from published and unpublished works by others.

Authors:

[illegible]

Zibran Zarif Amio
Student ID - 180041209

[illegible]

Fida Waseque Choudhury
Student ID - 180041215

[illegible]

Mohammed Mohaimen
Student ID - 180041217

Supervisor: [illegible]

Dr. Muhammad Mahbub Alam
Professor
Computer Science and Engineering
Islamic University of Technology

Co-supervisor: [illegible]

S.M. Sabit Bananee
Lecturer
Computer Science and Engineering
Islamic University of Technology

Acknowledgement

We would like to convey our sincere thankfulness to Almighty Allah for His favors, which have allowed us to complete this research. We would want to take this moment to recognise and thank all of the people who have helped us along the way and whose constant support, encouragement, and direction helped us produce this thesis.

We express our humble gratitude to our supervisor **Muhammad Mahbub Alam**, Professor, Department of Computer Science and Engineering, Islamic University of Technology (IUT). His supervision, guidance, depth of expertise, and unwavering support has been invaluable to us.

We are obliged to **S. M. Sabit Bananee**, Lecturer, Department of Computer Science, Islamic University of Technology (IUT). This research work would not see the light of success without his oversight, strategic ideas, and firm action. He has been a constant source of enthusiasm and encouragement which has allowed us to turn this endeavor into a success story.

Abstract

Cloud computing is hoped to replace traditional computing paradigms in the near future, as the Internet becomes a more integral part of our lives, more and more computing resources are being hosted in the cloud. One of the common techniques used by cloud service providers is to migrate cloud-based applications from one server to another for a variety of reasons. This thesis aims to add on the possible strategies of container migration in the cloud using QUIC in an innovative way. The idea is to use a dual-path extension of QUIC to ensure that the user's Quality of Experience is not hampered by the migration of the application hosted in the cloud server. This approach is coined as Dual-path in our thesis.

Cloud services are provided via containers that are processes running inside of the servers. Due to a number of conditions such as load balancing, resource balancing, hardware failure or maintenance etc. the container has to be migrated from one server to another. Traditional live migration techniques like Pre-Copy and Post-Copy consists of three rudimentary phases: iterative push phase, pulling of faulted pages and stop-and-copy (control transfer). During the control transfer phase the cloud service is unavailable and suspended until the container state is fully replicated to another target server. This introduces a downtime, hampering the end user's quality of experience. Furthermore, pulling faulted pages involves performance degradation which is not desirable. To mitigate the limitations identified in the traditional live migration techniques, we formulate the dual-path migration scheme.

Dual-path migration is an endeavor to redefine live migration techniques where an end user can simultaneously be connected to two servers at any given time. In this approach, once the migration is triggered the end user is dually connected to both the servers capable of requesting and receiving service from any of them. Initially, service is provided to the end user from the source server (traditional single path). Once the migration is triggered the container in the source server does not get suspended like the traditional schemes. Rather it will keep providing service to the end user and the transfer of control will be executed in the background. During this control transfer the end user can request data from any of the two servers. Since the end user is concurrently connected to both servers, the server having the requested data can respond. Once the background migration is completed it will simply terminate connection with the initial server and switch to the target server (again single path). The key attainment in this approach is its negligible downtime and performance upgrade. It also solves synchronization issues between the servers. In this work, we compare and contrast between traditional live migration techniques and our proposed dual-path migration by mathematically analysing post-copy migration using QUIC and dual-path migration, we show that under certain circumstances the dualpath migration scheme performs better than post-copy migration scheme.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Many technologies come and go, many ideas evolve and change over time. However, not all new things survive, some of them despite their ingenuity and beauty are not adopted on a wide scale, whereas other technologies quickly become famous, outshining all other contemporary rivals. This is true not only in computer science but also in many other fields too. Here are some of our thoughts (largely adopted from a talk given by Herb Sutter the ISO C++ Chair) on what makes a new successor technology gain widespread acceptance and adoption.

1. **Utility**
   All human inventions have flaws and holes which can be improved, all technologies have limitations and commonly known issues which demand resolution. So, any new replacement, if it aims to succeed, must not only provide the benefits of its predecessor technology but it must also add new features so that the overall benefit and power of the new technology outstrips all existing solutions. It must also aim to solve (as many as possible of) the old problems associated with the current technology so that deficiency of the old are supplanted by the new solution.

2. **Availability**
   In order to be massively deployed across the globe, the new solution must be able to adapt and not only shine in the previous domains where the old solution reigned supreme by Allah's will, but it must also be able to traverse new territory and domains in order to be widely adopted. The new solution must not be only applicable to a small subset of where the old solution was used, but rather it must be universal and general to all applicable fields.

3. **Adaptability**
   It must be remembered that the new solution is not the first solution in the world (well at least most of the time , it isn't), in order to replace the old solution, the new solution must be easily integratable into the existing infrastructure without any major changes. If the new solution requires restructuring of the existing prevalent system then that solution is not viable and cannot be deployed at mass scale due to the sudden paradigm shift it demands. The new solution must be able to co-exist with the existing solution and also must allow gradual incremental deployment. It is preferable that even partial application of the new solution yields benefits even though the rest of the system is running on the old solution. Moreover, the amount of effort required to implement the new solution should be minimal and

preferable proportional to the benefit it yields. Hence, people of all capacities and calibre can benefit by using the new solution according to their needs to improve and capacity to change.

4. **Upgradability**
The new solution is definitely not perfect (as any other human invention), hence must be open to modifications and extensions. There will be flaws and deficiencies in the new solution and also in the future circumstances are definitely going to be different as to what they are today and hence will require modification if the new solution wishes to survive in the long run. So, the new solution should keep the future in mind in its design plan and be flexible and agile enough to withstand the required updates smoothly by Allah's grace.

So, in this thesis we would like to use QUIC to implement server migration and we will show how QUIC encapsulates these features so we are hopeful that God Willing, our solution will be practically beneficial in real scenarios.

1.2 Server Migration and QUIC

QUIC represents the best effort by network engineers in the field of designing protocols above the network layer over the years. It aggregates many features of several previous protocols e.g. TCP, DTLS, RTP, SCTP and tries to take the best from each. An important feature about QUIC is that it is implemented in user-space hence is very easy to customise to add new extensions. The following table shows the ideas borrowed into QUIC from these protocols [1].

Table 1.1: Ideas borrowed into QUIC from protocols [1]

Feature	**Protocol**
Streams	SCTP
Application Data Unit	RTP
Running over UDP	DTLS, RTP
Multihoming	SCTP
Frames	SCTP
Security	TLS, DTLS

These features give a high score to QUIC on the utility requirement and it is also easily adoptable as it runs in user-space, meaning there is no need to upgrade OS to benefit from QUIC. This also means that QUIC is easy to upgrade so extensions to the protocol can be deployed easily simply through pushing software updates to deployed machines around the world.

On the other hand, server migration is the act of migrating a running server program from one machine to another machine. This is required for a number of reasons such as [2]:

- Mobile User: If the user is mobile e.g. on a car, then it might be in the best interest of the user that the server he is connected to migrates to a closer location in order to reduce latency.

- Administrative reasons : It might be the case that a certain machine is to be replaced or requires maintenance, in which case programs running on that machine needs to be migrated.
- Fault tolerance: Due to certain network failures a certain machine might be in a position where it has a single point of failure, in such cases the programs on that machine will have to be transferred to another machine which does not have a single point of failure.
- Load balancing: A certain machine may have too many programs running on it, so in order to conserve energy and reduce the load on the machine, some programs might have to be migrated to another machine.

1.3 Challenges in server migration

There are two types of server migration, cold (in which the program is suspended) and live (in which the program is running). Among the techniques of live migration three strategies are used, pre-copy, post-copy and hybrid. All three of them have advantages and disadvantages focusing on three parameters, namely total migration time, server downtime and performance during migration. In this thesis we work on post-copy strategy and contrast it with dual-path strategy as both are very similar in nature. Another challenge is that during migration, the client needs to establish a new connection with the new server by doing a new handshake. This adds extra time to the migration process and can be avoided with QUIC.

1.4 Solution by Integrating QUIC

QUIC has a feature known as connection migration which allows two peers which are in a session to change their IP address without breaking the connection. This is achieved as QUIC uses connection IDs to identify the connection instead of a pair of IP addresses and sockets. However, the QUIC specification [3], only specifies the client migration but not the server migration. Research is underway on implementing server migration with QUIC.

1.5 Contribution of the thesis

The main work presented in this thesis is to propose a new method of migration which leverages on the idea that the client will be simultaneously connected to both the old and new servers while the migration is ongoing (dual-path strategy). This idea is inspired by the multi-path extension of QUIC [4] in which the client can make parallel connections to the same server if the server holds multiple IP addresses to increase performance. We then mathematically compare dual-path strategy with the traditional post-copy strategy and prove that under certain circumstances dual-path outperforms post-copy.

1.6 Structure of the thesis

Introduction: Gives a brief overview of the subject matter along with the design points considered when choosing the solution.
Literature Review: Discusses the background material upon which this thesis is based and provides a detailed account of the related information already published in various journals.
Proposed Method: Compares and contrasts between the existing method and the proposed method after the improvements of the new method has been discussed. Also, a mathematical model is developed which analyses the two methods using probability theory.
Conclusion and Future Works: Wraps up the discussion and provides insights into the further investigation that is to be carried out to validate the thesis findings in real life.

Chapter 2

Literature Review

2.1 Background

Although QUIC was initially developed by Google (by Allah's permission), it has been standardised by IETF (Internet Engineering Task Force)[3], associated with QUIC is the new HTTP3 protocol which is the mapping of HTTP2 over QUIC.

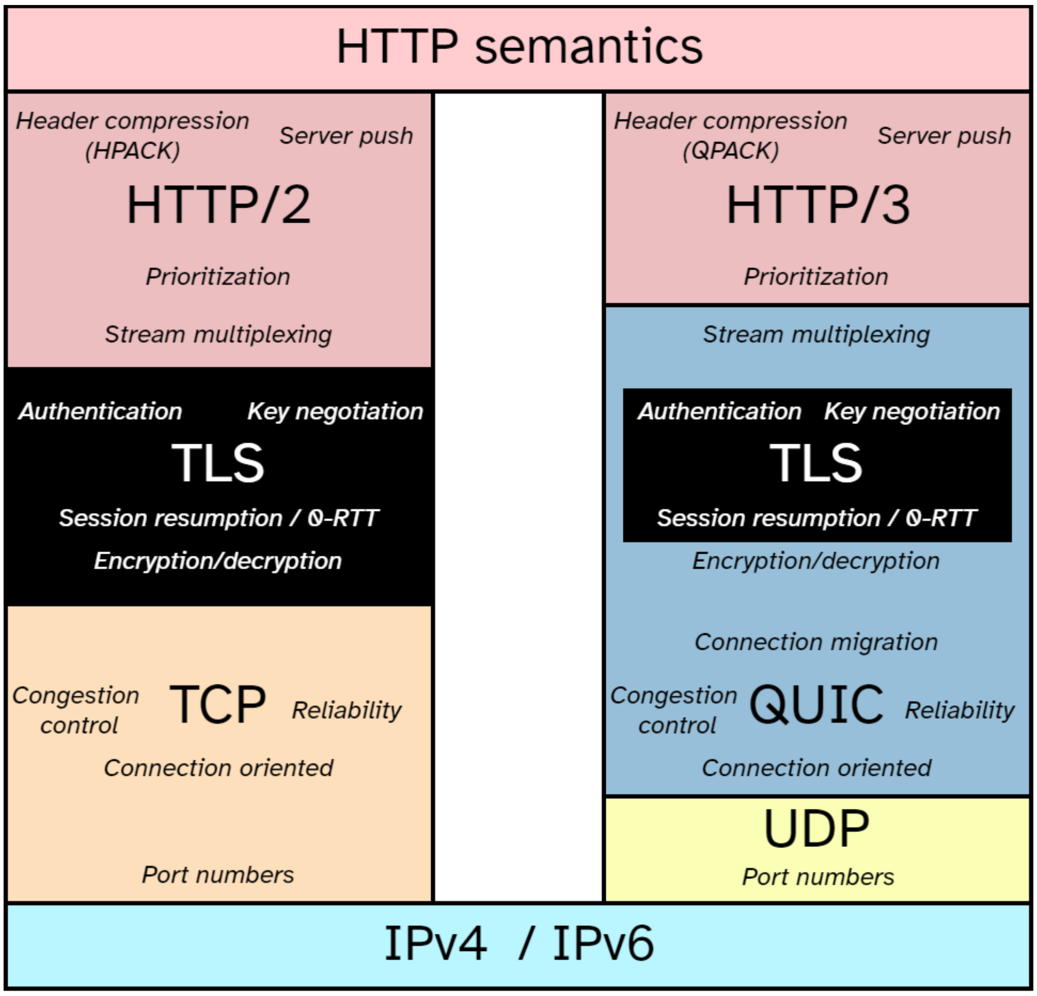


Figure 2.1: A comparison of various HTTP versions and QUIC (Source: Robin Marx (Reproduced by permission)

2.1.1 Overview of QUIC

Is QUIC an acronym?: Although, a quick internet search might reveal that QUIC stands for Quick UDP Internet Connections, I would not definitely assert it as an acronym as other sources clearly state that QUIC is a name by itself and is not an acronym.[5]

2.1.2 New Features of QUIC

The new features introduced by QUIC are:

1. **Connection ID**
   Unlike TCP which uses a combination of port-number and IP address to uniquely identify hosts, QUIC uses a combination of two numbers picked by client and server in order to identify each other. This has the advantage of enabling Connection Migration.

2. **Connection Migration**
   During a connection a host's underlying IP address may change due to multi-homing, being behind a NAT, moving from one network to another etc. In such scenarios the TCP connection is broken and a new connection needs to be established. In order to mitigate this problem, QUIC (by Allah's mercy) allows the connection to be resumed via Connection IDs.

3. **Reduced RTT Handshake**
   QUIC (by Allah's grace) reduces the handshake latency by 1-RTT by incorporating the TLS handshake with the transport layer handshake as compared to TCP where the TLS handshake occurs separately to the TCP

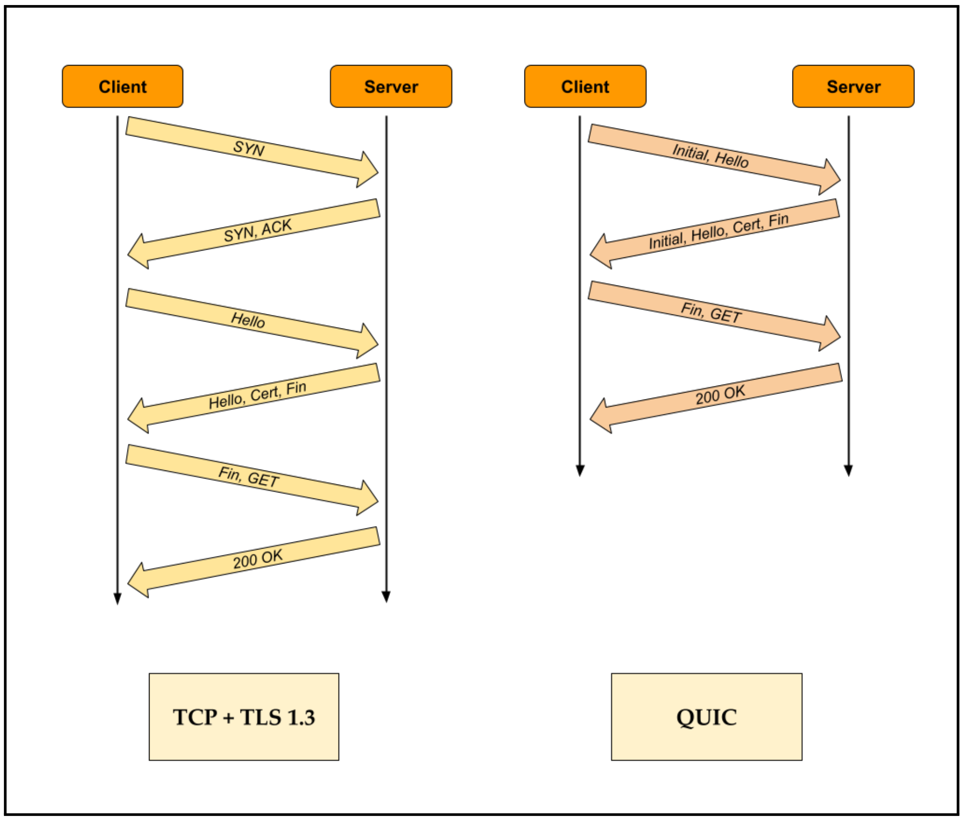


Figure 2.2: Comparison between TCP + TLS 1.3 and QUIC for first-time handshake [1]

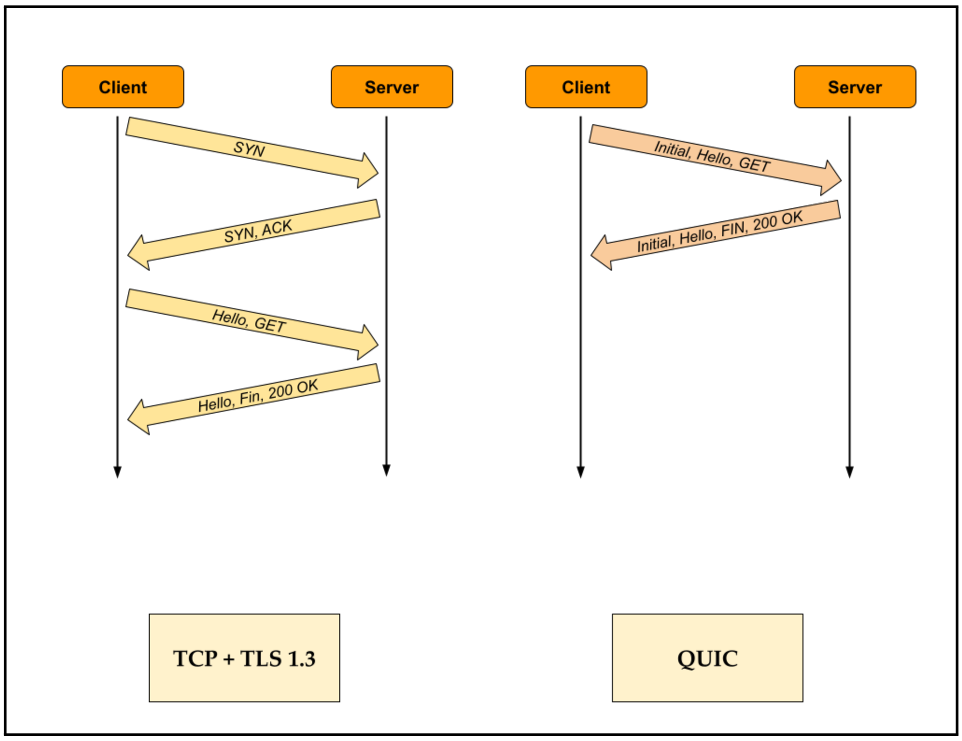


Figure 2.3: Comparison between TCP + TLS 1.3 and QUIC for next-time handshake [1]

4. **Running in User-Space**
Protocols such as TCP as usually part of the OS and are implemented in the kernel. However, QUIC is implemented in the user-space which allows rapid development and deployment without the need for OS upgrades (by Allah's permission).[1]

5. **Running atop UDP**
Middleboxes are notorious for tampering with protocols and modifying packets. Due to some intermediate middleboxes dropping unknown protocol's packets, new protocols may struggle to be deployed full-scale. So, QUIC uses UDP as a postman to overcome this hurdle (by Allah's permission) as UDP is already widely known and accepted.[6]

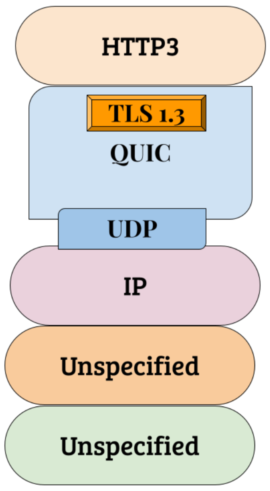


Figure 2.4: QUIC Network Stack

6. **Multiple Streams**
A stream can be considered as an "ordered sequence of bytes" as defined by Jana Iyengar of Fastly, in this sense TCP is a single stream connection between two hosts. QUIC implements multiple streams within a single connection with the purpose to solve the Head-of-Line-Blocking Problem of TCP. Streams can be unidirectional or bidirectional.[1]

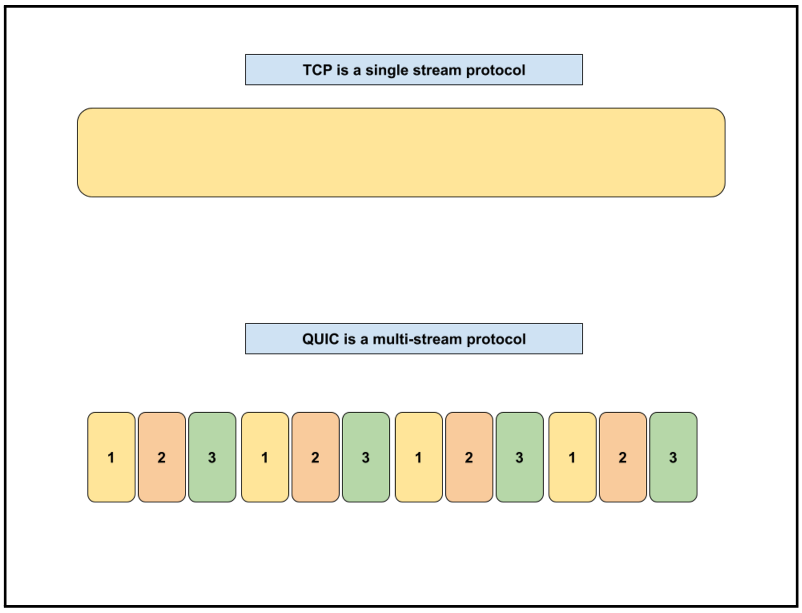

TCP is a single stream protocol
QUIC is a multi-stream protocol
1
2
3
1
2
3
1
2
3
1
2
3

Figure 2.5: TCP sends all the data as a single byte stream, but QUIC sends data where each data is associated with a separate stream. This particular example shows round-robin scheduling with 3 streams [1].

7. **Stream Multiplexing**
   As there are multiple streams, the question arises as to how these streams should be multiplexed together and sent across the single connection 'pipe'. Various multiplexing techniques could be used such as round robin, first-come-first-serve and so on.

8. **Packet Format**
   The packet format for QUIC is very different from TCP. Each packet has a monotonically increasing packet number, along with the packet header and the payload. In payload, there are multiple frames which carry different types of information depending on the **frame type** e.g. CRYPTO frames, ACK frames, STREAM frames. The STREAM frames are data-carrying frames and each frame has (among other fields) a unique stream ID, the byte offset within that stream and the actual data. A bunch of frames are packaged together into a single UDP datagram for transmission.

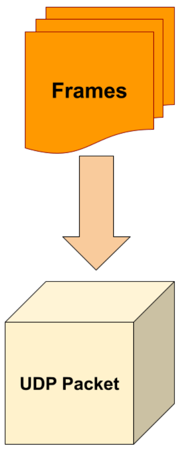


Figure 2.6: Packaging of QUIC frames into UDP Packets

9. **Frames**
   Every QUIC packet consists of a bundle of multiple frames, some frames carry data whereas other frames are only commands to the peer to perform any action. Frames can vary in size and dimension, and they can be categorised into groups depending on their functionality. New Frames can also be defined by protocol extensions to enable new functionality and improve the functionality of the QUIC protocol. There are approximately 20 different frames, they can be grouped into several categories:

   - Data – STREAM (contains stream data)
   - Connection – CONNECTION_CLOSE
   - Extension (for adding new features) – SERVER_MIGRATION
   - Probing – PADDING, PATH_CHALLENGE, PATH_RESPONSE
   - Control – PING, ACK, STOP_SENDING
   - Cryptographic – CRYPTO
   - ..and more

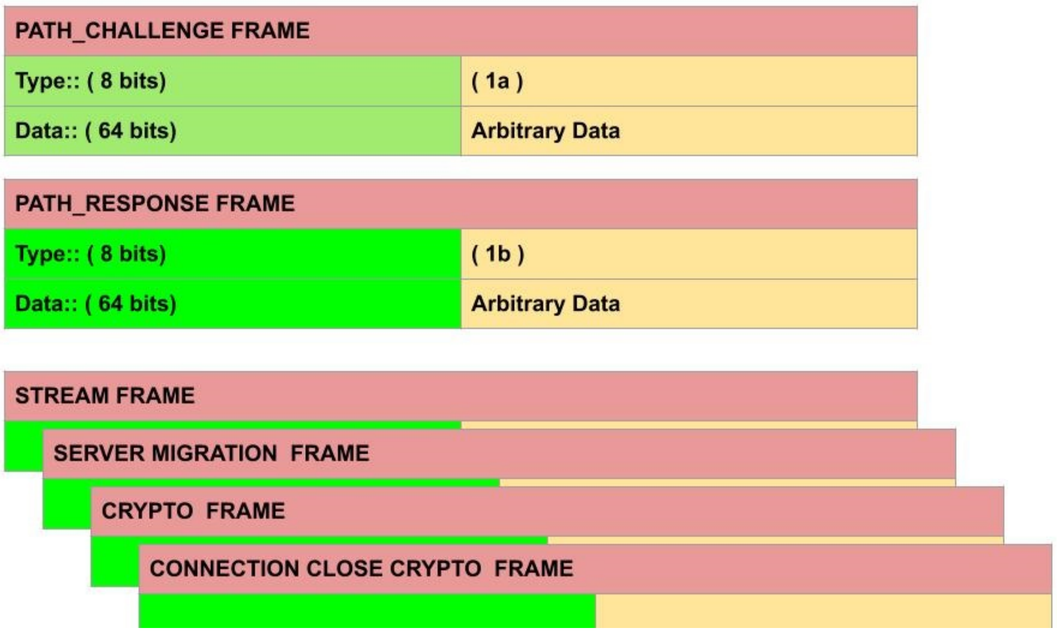


Figure 2.7: Some Frames in QUIC [1]

2.1.3 Before QUIC

Transport Control Protocol (TCP) is the most widely used transport layer protocol due to its reliability and all the services it provides. It is very good at forming a pipe between the sending and receiver through which packets can be reliably sent. The intermediary nodes, routers, firewalls, etc are all very familiar with this protocol. It is often used with TLS for secure transmission of data.

TCP is a process to process protocol. It uses port numbers to form a virtual connection between two TCPs, and is therefore a connection oriented protocol. This allows TCP to use flow control and congestion control mechanisms at the transport layer level. It sends data as a stream of bytes. There are buffers at the sender and receiver to make sure congestion and flow control can be done effectively.

TCP is reliable as it uses the processes of sending acknowledgements and retransmissions to make sure the data is sent reliably and lost packets are resent. Data must be sent and received in order so out of order packets don't happen. The receiver acknowledges for the packets it has received, and if a packet is lost it is retransmitted. This is done either when the timer runs out, or when three duplicate acknowledgements are received.

TCP implements flow control by giving the receiver the duty to determine the rate of data sent. This makes sure that the receiver is not overwhelmed with data. It uses a byte oriented flow control. A sliding window technique is used which is also byte oriented. It also uses a mandatory checksum for error control and making sure the data was not corrupted.

TCP acknowledges congestion in the network, and can use methods like Slow Start, Additive Increase Multiplicative Decrease (AIMD), etc to deal with congestion in the network. Collision avoidance and collision detection algorithms are used for this. There are also different types of TCP like TCP Reno, TCP Tahoe, etc which use slightly different methods to achieve congestion control.

2.1.4 Comparison between QUIC and TCP

Before discussing the differences it is significant to elaborate on the similarities:

- **Reliable**
  Both TCP and QUIC guarantee packet transfer to it's peer by checking Acknowledgements and retransmitting in case of packet loss.

- **Secured with TLS**
  Both TCP and QUIC are cryptographically secured via cryptographic protocols such as TLS. So, the transmitted data is encrypted and is not visible to onlookers.

- **Host-host connection**
  Both TCP and QUIC are only concerned about the final endpoint and are not responsible for the intermediate routers which forward the packets between endpoints. Neither TCP nor QUIC handles packet routing or any functionally of the network layer.

- **Flow controlled**
  In order not to overwhelm the receiver, both TCP and QUIC is flow control mechanisms to only send data at a date which can be handled by the receiver.

- **Congestion Controlled**
  In order not send data beyond the capacity of the network both TCP and QUIC use congestion control to limit the amount of data send into the network. However, just like TCP, QUIC does not have any fixed congestion control mechanism, so any standard congestion control mechanism e.g. NewReno, Cubic, BBR or custom ones could be used.

- **Ordered byte stream delivery**
  Both TCP and QUIC offer a delivery promise to the application layer that the data will be sent to the peers application layer in the same order as the source has specified by passing it down to its transport layer.

- **Handshake and teardown mechanisms**
  Both TCP and QUIC have connection setup (handshaking) and also connection teardown mechanism built into them. So, both the start and the end of connections are clear and unambiguous [7].

Table 2.1: TCP vs QUIC

TCP	**QUIC**
Uses a combination of port-number and IP address to uniquely identify hosts	Uses a Connection ID to identify hosts
Connection Migration is not possible	Connection Migration is possible
TLS handshake occurs separately to the TCP handshake	TLS handshake is incorporated with the transport layer handshake, reducing the latency
It is part of the OS and is implemented in the kernel	It is implemented in the user-space which allows rapid development and deployment without the need for OS upgrades
TCP is a single stream connection between two hosts	QUIC implements multiple streams within a single connection with the purpose to solve the Head-of-Line-Blocking Problem of TCP

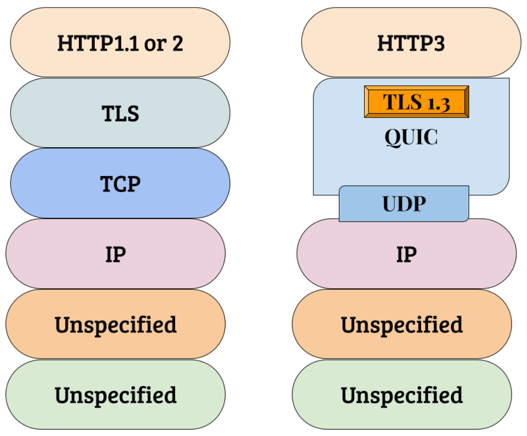


Figure 2.8: QUIC stack vs TCP Stack [6]

The QUIC protocol runs atop UDP, the main reason being that protocols other than TCP or UDP are blocked by middleboxes fearing security threats. This has hampered the deployment of new protocols in the past, so QUIC tries to avoid this problem entirely by appearing to these middleboxes as UDP packets to pass through their filters. QUIC also integrates TLS1.3 closely to its handshake protocol inorder to reduce the number of round-trip-times it takes to setup the connection.

2.1.5 Overview of HTTP 1, 1.1, 2 and 3

HTTP 1.0 was the first iteration of the HTTP protocol officially introduced in 1996. It was good for being the first protocol but soon a lot of problems and inefficiencies were detected which caused the introduction of HTTP 1.1. HTTP 1.1 aimed at solving some of the issues with HTTP 1.0.

HTTP 1.1 introduced persistent connections, meaning multiple requests/responses can be made in a single connection. For HTTP 1.0, a new connection had to be opened for each pair of request and response. That caused a lot of delay due to things like TCP Slow Start.

HTTP 1.1 also introduced the OPTIONS method which allowed the client to determine the abilities of the server it was communicating with. Beside that, the new protocol expanded on the limited caching abilities of HTTP 1.0; new status codes; compression support, etc.

However, in 1997, when HTTP 1.1 was introduced, there was not much media, and most web pages were static. There was also HTTP head of line blocking, where the client limited the number of connections per host (usually 6). As websites started including more and more media, most of them started having lots of objects, some closer to 200. This caused a lot of inefficiencies. There was also repetition of header data. Some workarounds were available, like sharding, spriting, bundling, etc. but they were not enough to keep up with the growing size of the internet and web pages [8].

Therefore, in 2015, HTTP 2 was introduced. It used a single connection per host, but within that connection lots of parallel streams could be set up. That solved the HTTP head of line blocking problem, but the TCP head of line blocking issue was still there. For example, if there was a single connection with a 100 streams, sending 100 parallel images, if 1 packet is lost, all the streams have to wait for the retransmission of the lost packet.

That called for a new protocol to fix the inefficiencies. However, by then, there were already a lot of middle boxes who knew what to do, and replacing them was not practical. This caused a problem known as ossification. Intermediary routers, gateways, etc run software to handle network data. They upgrade much slower than the edges. This prevented network innovations like TCP improvements (TFO for example), TCP/UDP replacements, new compression algorithms (like HTTP broti) [9].

That is why HTTP 3 was introduced, which works with QUIC protocol. It was initially deployed by Google, but in 2015 IETF started working on it. Now it has one transport protocol (QUIC) and one application protocol (HTTP3). This fixed TCP head of line blocking issues. Here, only the affected streams have to wait, and other streams can continue sending - making the streams independent. The handshakes are faster due to the incorporation of TLS into the transport layer handshake. Earlier sending of data is also possible with HTTP 3. For example if index.html is requested, the server can send the css and javascript files as well. HTTP3 header compression scheme QPACK is also different from HTTP2's HAPCK due to the independent streams available in HTTP3.

2.1.6 Overview of TLS 1.2 and 1.3

TLS was designed to provide cryptographic security to transport layer protocols like TCP or QUIC. It is sandwiched between TCP and the Application layer. The Latest TLS versions, like TLS 1.2 and TLS 1.3, consist of multiple protocols. Handshake protocol uses digital certificates to authenticate client and server, then exchanges necessary key shares for symmetric encryption during data transmission. The record protocol ensures data integrity via MAC (Message Authentication Codes) and confidentiality by advanced encryption techniques like, AES, CHACHA etc. Alert protocol Handles different warning and error messages between the Client and the Server. TLS 1.2 is the most widely used and secure version of TLS with around 99% website support. TLS 1.3 is gaining popularity in recent times with more than 50% of websites supporting it. It is the fastest and the most secure version of TLS which is immune to most known threats. Any version prior to TLS 1.2 is deprecated due to security threats. The handshaking comparison is depicted in figure 2.9.

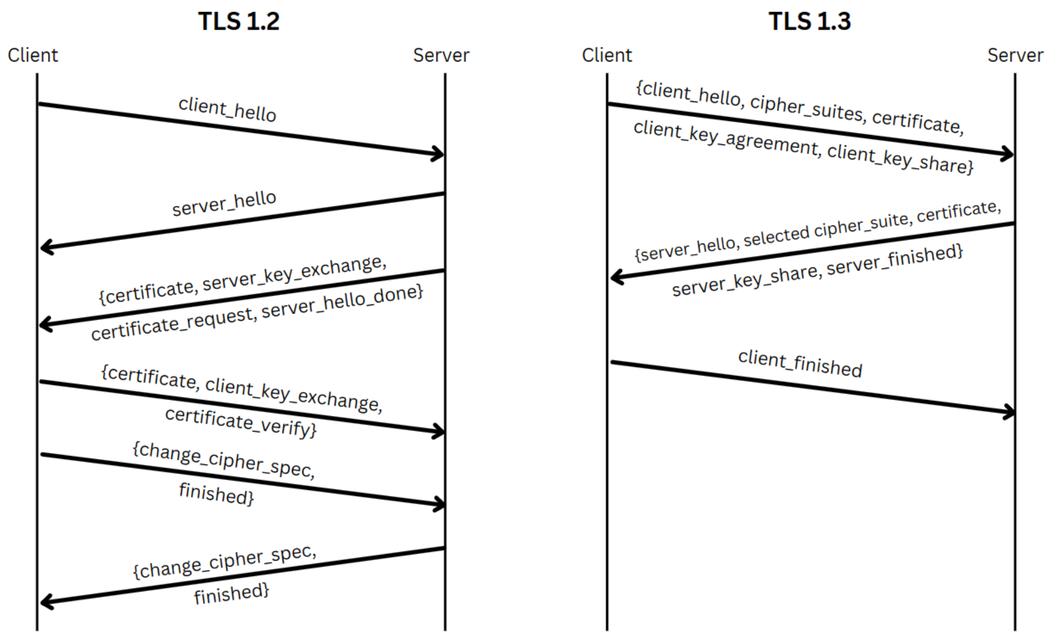

TLS 1.2
Client
Server
client_hello
server_hello
{certificate, server_key_exchange, certificate_request, server_hello_done}
{certificate, client_key_exchange, certificate_verify}
{change_cipher_spec, finished}
{change_cipher_spec, finished}
TLS 1.3
Client
Server
{client_hello, cipher_suites, certificate, client_key_agreement, client_key_share}
{server_hello, selected cipher_suite, certificate, server_key_share, server_finished}
client_finished

Figure 2.9: Handshake comparison of TLS 1.2 ans TLS 1.3 [10]

Table 2.2: TLS 1.2 vs TLS 1.3

Aspect/Feature	**TLS 1.2**	**TLS 1.3**
Total RTT required to complete handshake	2-RTT	1-RTT
Latency to establish secure connection	Relatively more	Relatively less
Security	Less secure	More secure
Number of the combinations of cryptographic algorithms available	37	5
Algorithm used for certificate verification	Typically RSA	Always Diffie-Hellman
Perfect Forward Secrecy	Not available	Available
Algorithm used for asymmetric key exchange	Typically Diffie-Hellman	Always Diffie-Hellman
Session resumption (0-RTT)	Not available	Available

2.1.7 Overview of TLS 1.3 Integrated into QUIC

Connection-oriented protocols like TCP use a separate TLS layer on top, typically TLS 1.3, that provides peer authentication, data confidentiality and integrity. QUIC requires TLS as well. But it is not fully dependent on it. TLS does not work separately from QUIC. For authentication and parameter negotiation purposes, QUIC relies on TLS 1.3. But for reliability, ordered delivery and record layer functions like confidentiality and integrity TLS depends on QUIC. So, instead of strict layering, both QUIC and TLS cooperate to provide a secure and reliable channel. TLS provides the handshaking mechanism to QUIC. In contrast, QUIC provides a mechanism to carry the data ensuring both confidentiality and integrity. Basically, QUIC does the job of TLS record layer. The handshaking feature is identical to TLS 1.3 with added QUIC reliability considerations. Like TLS 1.3 handshaking, client and server communicate via messages like client_hello, server_hello etc. Then, these messages are wrapped inside of QUIC frames like CRYPTO frame, STREAM frame etc [10]. These frames are again wrapped inside of QUIC packets which are actually transmitted. There are usually four types of QUIC packets, Initial, 0-RTT, Handshake and 1-RTT each having their own encryption levels. If multiple packets are needed to be transmitted then all of them are packaged inside of UDP datagrams. Considering when data is being transmitted by the client, there are two handshaking modes in QUIC that are enabled by TLS 1.3:

Full 1-RTT handshake
The client is able to transmit application data after one round trip. This happens during the first time connection establishment with the server where both client and server have to negotiate on security parameters. The flow of 1-RTT handshake is as follows:

- Client sends an Initial packet which contains a CRYPTO frame. Inside this CRYPTO frame is the client_hello.
- Server replies back with an initial packet which has an ACK frame and a CRYPTO frame containing server_hello message. It also sends a Handshake packet contain-

ing server certificate, server key share etc. The server can optionally send a 1-RTT packet with STREAM frame that contains application data.

- Client sends an Initial packet that has an ACK frame. It also sends a Handshake frame with a CRYPTO frame having the finished message. At this stage, the client can send application data wrapped inside a STREAM frame within a 1-RTT packet. So, exactly after one round trip the client can send application data.
- Finally, the server replies with a Handshake frame containing an ACK frame and a HANDSHAKE_DONE message inside a 1-RTT packet.

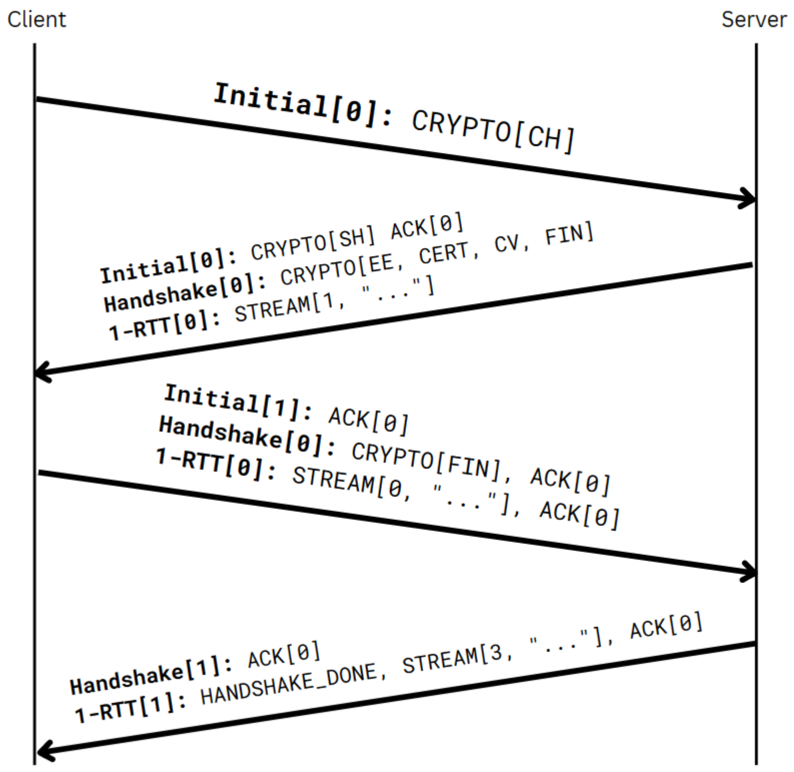

Client
Server
Initial[0]: CRYPTO[CH]
Initial[0]: CRYPTO[SH] ACK[0]
Handshake[0]: CRYPTO[EE, CERT, CV, FIN]
1-RTT[0]: STREAM[1, "..."]
Initial[1]: ACK[0]
Handshake[0]: CRYPTO[FIN], ACK[0]
1-RTT[0]: STREAM[0, "..."], ACK[0]
Handshake[1]: ACK[0]
1-RTT[1]: HANDSHAKE_DONE, STREAM[3, "..."], ACK[0]

Figure 2.10: QUIC-TLS 1-RTT Handshake [11]

0-RTT Handshake

In this case, the client uses previously negotiated security features and can send application data in the very first flight. The process is exactly the same as 1-RTT handshake, with only one change, i.e, the client will send a 0-RTT packet that has a STREAM frame carrying application data on the first transmission. It doesn't have to wait for one round trip.

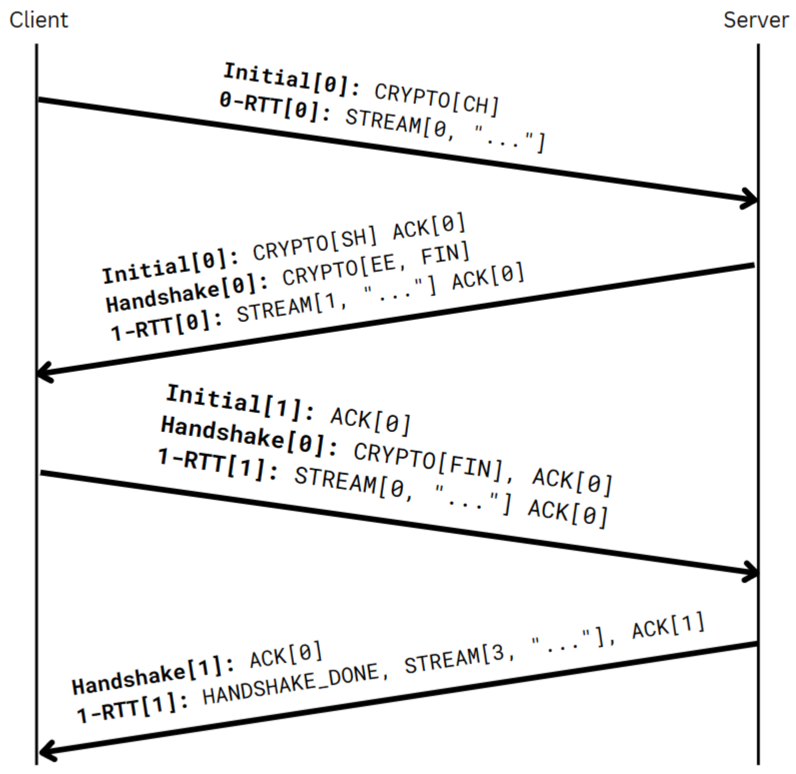

Client
Server
Initial[0]: CRYPTO[CH]
0-RTT[0]: STREAM[0, "..."]
Initial[0]: CRYPTO[SH] ACK[0]
Handshake[0]: CRYPTO[EE, FIN]
1-RTT[0]: STREAM[1, "..."] ACK[0]
Initial[1]: ACK[0]
Handshake[0]: CRYPTO[FIN], ACK[0]
1-RTT[1]: STREAM[0, "..."] ACK[0]
Handshake[1]: ACK[0]
1-RTT[1]: HANDSHAKE_DONE, STREAM[3, "..."], ACK[1]

Figure 2.11: QUIC-TLS 0-RTT Handshake [11]

2.1.8 Comparison between TCP-TLS and QUIC-TLS

TCP is a connection-oriented transport layer protocol that creates a bidirectional tunnel between two processes for reliable transmission of data. Within this tunnel TCP splits the data into packets, acknowledges them, assembles them into the right order, removes duplicates and retransmits packets that are lost. These packets of data are vulnerable to security threats like unauthorized access or data alteration. So, TCP uses a separate TLS layer on top, typically TLS 1.3, that provides peer authentication, data confidentiality and integrity. The QUIC protocol is also a transport layer protocol that sits on top of UDP, a connectionless protocol that doesn't guarantee reliability. However, QUIC protocol is designed to provide identical services as that of TCP. It also uses TLS 1.3 for cryptography and security purposes. The primary difference between how TLS is implemented in TCP compared to QUIC can be noticed in their architecture as depicted in figure 2.1. In the case of TCP, TLS is a fully separate layer. The TLS handshake occurs only after TCP is done with connection establishment via the three-way handshaking (SYN, SYN+ACK, ACK). In contrast, QUIC implements TLS as an essential component within itself. In QUIC, TLS does not perform separate handshaking to establish security parameters. Rather, QUIC and TLS work hand in hand to establish both a reliable and cryptographically secure connection at the same time. So, QUIC reduces 1-2 RTT depending on the version of TLS (1.3 or 1.2), by achieving the outcome of performing both handshakes in one go [12].

In the case of TCP, the TLS consists of a record protocol. This protocol splits any data into units called records. A record has a certain field for indicating the type of data being transferred over the secure channel like whether its alert, handshake or application data etc. Then there are other fields concerning data lengths, protocol versions etc. So, the record protocol, in TLS, carries any type of data within the channel ensuring integrity and confidentiality. QUIC takes over this responsibility of the record layer and replaces it from TLS [13]. For example, QUIC uses CRYPTO frames to carry TLS handshake data. These frames are encapsulated in QUIC packets. Instead of record layer encryption, QUIC packet protection is used to protect handshake data. TLS only provides a secret, an Authenticated Encryption with Associated Data (AEAD) function and a Key Derivation Function (KDF) to QUIC for data encryption. Similar to CRYPTO frames, STREAM frames are used to carry TLS application data. TLS alerts are also converted to error codes called CONNECTION_CLOSE. When multiple QUIC packets are needed to be sent, all of them are coalesced in the same UDP datagram [14].

To protect corresponding type of data, QUIC obtains the encryption levels and its keys from TLS based on the packet types (Initial, 1-RTT, Handshake, 0-RTT). The client uses the CRYPTO frame to provide an inchoate client greeting (CHLO) message to the server during the first connection initiation [15]. The server replies by sending a reject (REJ) message that includes information about the server settings, server certificate, source-address token, etc. In later interactions, the client's source-address token is utilized to confirm their identity. The client builds a complete CHLO message from the data in the REJ message, which it then transmits back to the server. The Diffie-Hellman key sharing for the client is contained in this message. The server responds by sending a Server Hello (SHLO) message, signifying a successful handshake [16]. This is the process of 1-RTT handshakes. To achieve 0-RTT for future connections, the client includes NewSessionTicket messages in the CRYPTO frames which indicate 'early_data' allowing the client and server to form a connection with previously agreed parameters. Here, the client can actually transmit STREAM frames containing application data on the first flight. TLS 1.3 with TCP would use PSK (Pre-Shared Keys) to achieve 0-RTT and session resumption [17].

Table 2.3: TLS 1.3 with TCP vs QUIC [4]

Aspect	**TLS 1.3 with TCP**	**TLS 1.3 with QUIC**
Architecture	TLS is a standalone protocol separate from TCP	TLS is a feature/part of QUIC
Record protocol	Record protocol carries TLS data	QUIC protocol does the job of record protocol
Data Carriage	Records are used to carry any type of data within the secure channel	Data specific frames are used to carry only that type of data, e.g. CRYPTO frames for TLS handshake data
Error messages	TLS Alert protocol notifies the peers about protocol failures	QUIC uses CONNECTION_CLOSE error codes
0-RTT	Achieved with PSK	Achieved with NewSessionTicket messages in the CRYPTO frames
TLS versions	Older versions of TLS can be integrated like TLS 1.2 or 1.1	Any version older than TLS 1.3 is not compatible. Newer versions can be integrated based on both client and server's ability to support.
Key Update	Keys are updated with ChangeCipherSpec or KeyUpdate mechanisms	QUIC packet protection does the job
TLS Middlebox Compatibility Mode	Is used	Is not used
Vulnerability to replay attacks	Vulnerable	Not vulnerable

2.1.9 Multipath QUIC

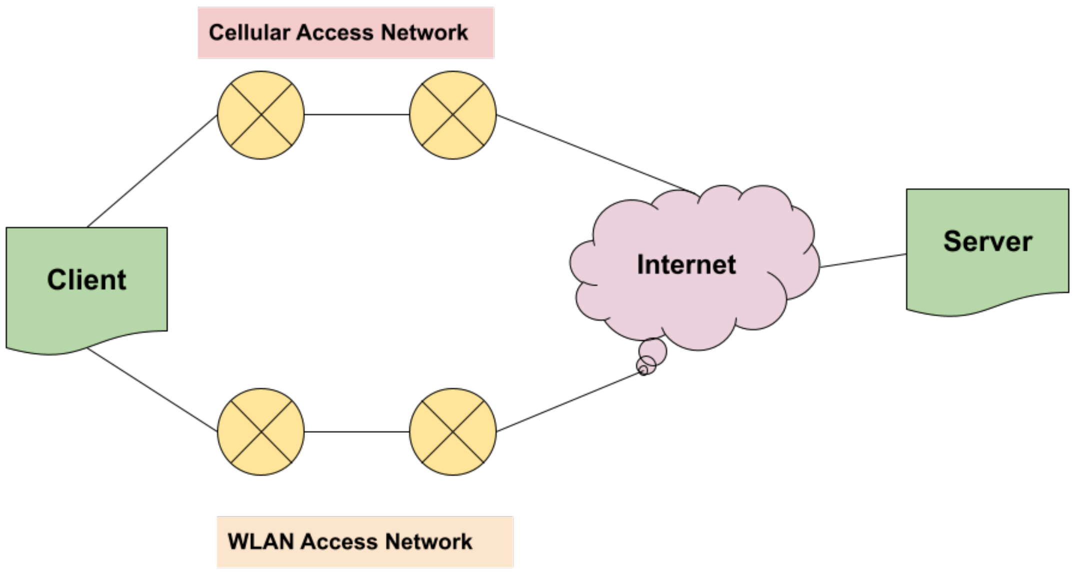


Figure 2.12: Multipath Connections of a smartphone

The multipath extension to QUIC is a working draft at the IETF [18]. The basic idea of multi-path QUIC is that an end-point can communicate with its peer via more than more path (which might not necessarily be completely disjoint). Suppose that a smartphone has both Wifi and cellular data available. The user might wish to download files using both Wifi and cellular data in order to leverage the additional bandwidth with the aim to reduce the download duration.

In multipath QUIC, an endpoint informs its peer about another IP address at which it is also available, the peer then validates the new address by sending a PATH_CHALLENGE and expects to receive a PATH_RESPONSE from the same address. Thereafter, the endpoint is able to send data and also receive acknowledgements via both paths to its peer. The PATH_CHALLENGE contains encrypted arbitrary data, the peer upon receiving this data echoes back the same data.

2.1.10 Edge Computing

Internet of Things (IoT) devices generate enormous amounts of raw data, typically measured in zettabytes, in various formats and at a very high rate. It is known as Big Data. To store, process and analyze Big Data, Cloud Computing emerged as a solution providing elasticity, scalability, security and accessibility. However, in Cloud Computing, all these complex sets of data are stored and processed in data centers that are located at a distance far away from the original data generation point causing bandwidth and latency issues. Edge Computing was therefore developed as a layer in between end users'/devices' and cloud data centers. By processing and analyzing the created data while relocating to the edge, which is the closest position to the original source point, it serves as an optimization to the cloud computing architecture. With less traffic and less maintenance required, edge computing offers extremely low latency, high bandwidth, and continuous connectivity [19]. It also improves scalability and reduces redundant cost by operating only on the relevant part of data. The three-level architecture of Edge Computing is depicted in figure 2.13.

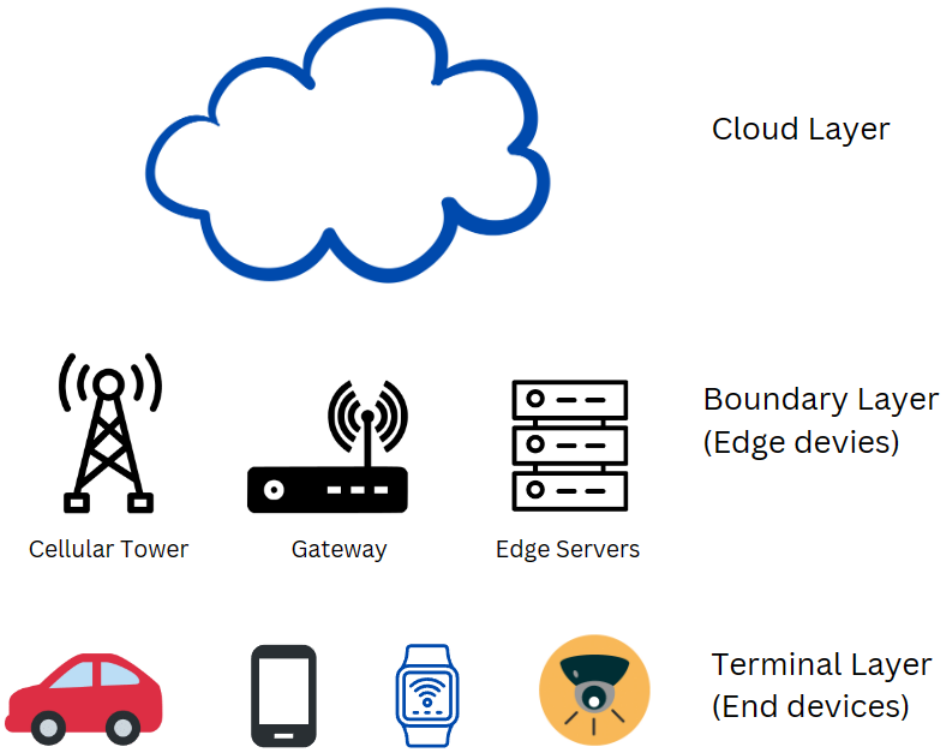


Figure 2.13: Edge computing three level architecture [20]

Terminal Layer: It consists of the IoT devices like sensors, drones, intelligent cameras, smartphones and smart watches, vehicles etc. These devices are directly associated with Big Data. The generated data is processed by the Edge/Boundary layer.

Boundary Layer: This layer consists of the edge devices. Examples of edge devices are Wi-Fi or 4G/LTE cellular networks and mobile internet. Sensors can transmit data via Wi-Fi routers and switches, smartphones can use mobile internet to forward raw data. Edge devices can process data and communicate back and forth with the end user by itself without the intervention of the cloud. It only acts as a middleman or interface for the cloud. To perform complex operations it may redirect the tasks to the cloud.

Cloud Layer: It is the layer with cloud computing itself. Cloud Computing provides all the services associated with storage and data analytics [21].

2.1.11 Container vs Virtual Machine

Virtual machine or VM, typically called an image, is a software-defined computer system that exists only as code, but behaves like a real computer. Physical computers are tangible and have dedicated hardware resources. VMs borrow these resources to virtually create CPU, network interfaces and disk of their own providing identical services to that of a physical one. The physical machine, in this case, is called a host and the VMs running on top of the host are called guests. VMs are administered by a software called Hypervisor, that separates host hardware resources from the guests and provisions them to each existing guest VM appropriately. Recently, like VMs, Containers are gaining momentum with the introduction of systems like Docker. It is a platform for creating, deploying, and managing containerized applications. The main distinction between containers and virtual machines is that with containers, all of the guests share the same host operating system while with virtual machines, the host operating system is distinct from the operating systems of the running guests. The architectural difference is depicted in figure 2.14. The advantages of Containers are [22]:

- Since Containers share the same OS with the host, they are very lightweight and highly portable compared to VMs which each have isolated OSs.
- Containers can boot up much faster than VMs. So they can be switched on and off within seconds reducing maintenance overhead.
- Containers are more application-centric, whereas VMs are considered to be a standard package having all the features. When similar applications require a single OS kernel then Containers perform better.
- Resource and memory consumption is lesser in Containers achieving low redundancy.
- Sharing of files is possible between Containers which VMs don't support.

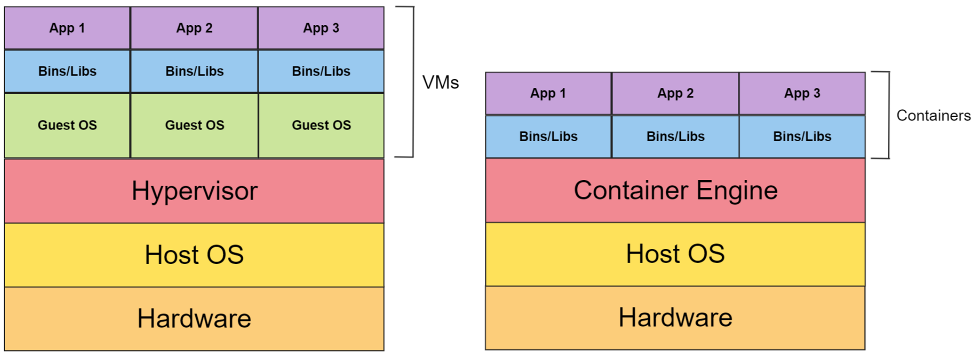


Figure 2.14: Virtual Machine vs Containers [23]

Containers and VMs consolidate servers and better utilize hardware resources. They can run separate processes in isolated environments and provide scalable on-demand services with improved data security and disaster recovery. One of the major features of technologies like Containers is its portability and migration. A Container must reduce performance overhead, achieve energy efficiency, and achieve load balancing in order to offer the best service to the client. To do so, a VM often has to move across different hosts. This is known as Container migration from one host to another. Container migration must be done in a seamless manner without interrupting the applications or services running inside the Containers. When moving a Container from one host to another inside the same LAN, the CPU state and memory pages are copied, and the Container's disk can be mounted on a shared medium that is reachable from the destination. The Container's disk, together with the memory and state, must be copied when migrating to a destination host situated in another LAN that is connected via WAN. The least amount of data should be duplicated during Container migration, with the least amount of interruption and delay. All network connections and traffic to the Container are redirected to the new location when it has been transferred to the destination and started running there.

2.2 Related Work

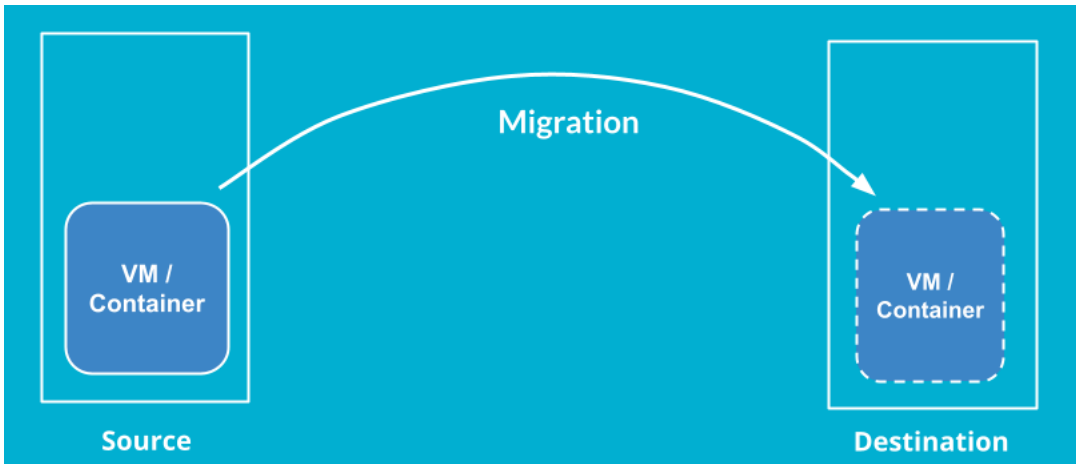


Figure 2.15: The main idea behind migration

The current version of the QUIC i.e.v1 only specifies the details for client side migration. It also mentions that anyone implementing server side migration ought to secure their mechanism against request forgery attacks [3]. However, not one of the existing works which I have seen actually address any security analysis nor any mention of the security aspects of their mechanisms.

The main work on providing server side migration support in QUIC has been done at the University of Pisa, Italy, the primary work[24] and a thesis[25] which proposes two new mechanisms for server side migration.

The main idea in the primary work is that the client is informed of the new address to which it has to migrate beforehand by the server, and when the client's packets do not receive further acknowledgements from the server (as it has migrated to a new IP address), the client tries to reach the server at the new address(es). Thereafter there is an exchange of path validation packets between them until the client can resume receiving service from the server as usual.

In the thesis, the author further extends upon these mechanisms and instead of the client having to probe the server at the new-address the server actually informs the client after migration and the client then sends a PATH CHALLENGE frame and the server then responds with a PATH RESPONSE frame to prove it's validity. Thereafter the client and server can resume normal operation. [26]

The following are some key parameters that are used to assess and compare the performance of migration processes:

Total Migration Time (TMT): It is the amount of time it takes between the beginning of the migration process at the source host and the point at which the container is fully migrated and operational at the destination.

Downtime (DT): The period of time between the Container's suspension at the source and its restart at the destination is when the Container state is typically copied. Downtime is the total time the service is unavailable, so it should be kept as minimum as possible.

Total Transferred Data (TTD): The amount of data moved during a migration directly affects the duration of the migration and any downtime. The amount of data transferred is directly proportional to the overall bandwidth utilization. Stateless migration and stateful migration are the two possible types of container migration based on the preservation of container state.

Stateless Migration: A new Container is produced at the destination after the Container in the source host is deleted. It causes the loss of Container state.

Stateful Migration: Before the Container in the source host is suspended, the state of the Container—including its CPU status, memory pages, registers, disk contents, etc.—is transferred to the destination host. The Container doesn't restart in this scenario; instead, it resumes from its previous state. The state is preserved. Stateful migration can be classified into Cold or non live migration and hot or live migration [27].

Cold Migration: The Container remains suspended for the entire duration of migration. It works as follows:

- First it suspends/freezes the Container at the source host.
- Then it checkpoints the last snapshot of the Container state and starts migration. During this time no instance of the Container is running.
- Finally, after the migration is complete the Container resumes at the destination.

Although cold migration provides a simple solution to Container migration, it causes long downtimes which is proportional to total migration time.

Live Migration: It is the process of moving containers without stopping service, minimizing the amount of downtime caused by container suspension. Live migration has emerged as a vital technique in cloud management. The migration process is more complex compared to cold migration but reduces downtime to almost zero. Considering where the Container is running and data being transferred, live migration consists of three particular phases or mechanisms: push phase, stop-and-copy and pull phase [28].

Push Phase: The Container is active on the source host and serving the client directly during this phase. As seen in figure 2.16, the source host, at the same time, pushes every memory page to the destination.

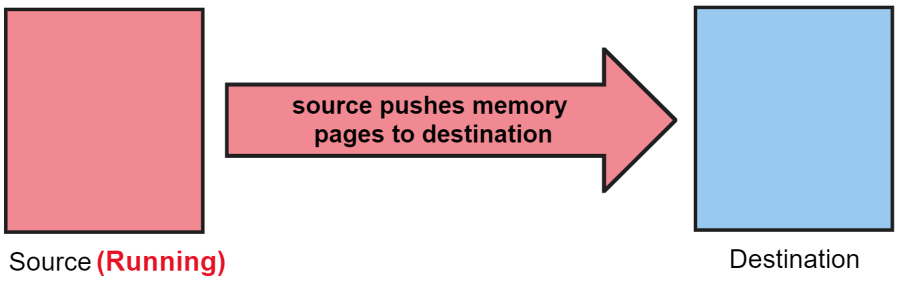


Figure 2.16: Push phase

Stop-and-Copy: Control is moved from the source host to the destination host during this phase. The Container is initially stopped on the source system, and its contents—including its state, memory pages, and disk data—are then moved to the target host. The Container at the destination host is initiated at the termination of this phase.

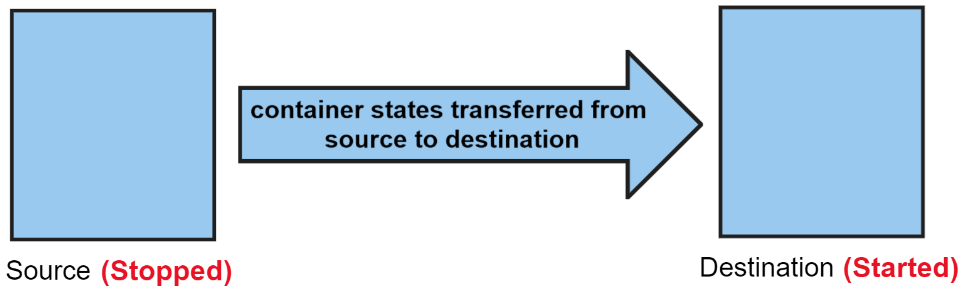


Figure 2.17: Stop-and-copy phase

Pull Phase: During this stage, the container is operating on the destination host and serving the client from there. The destination host also pulls the necessary memory pages from the source host at the same time.

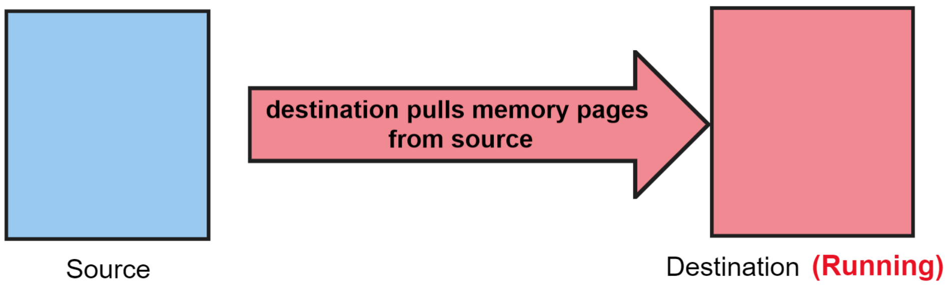


Figure 2.18: Pull phase

Based on these three phases there are three different live Container migration techniques: Pre-Copy, Post-Copy and Hybrid migration.

Pre-Copy: It is based on stop-and-copy and the push phase. Memory pages from the source host are repeatedly transmitted to the destination host during the push phase. The first iteration checkpoints the current state of the Container at the source and transfers the associated memory pages to the destination. The Container state and the memory pages are changed as a result of the writing of new data every second. The pages that were transferred during the first iteration are therefore no longer identical to the original pages. The term "dirty pages" refers to these altered memory pages. The filthy pages are moved to the destination during the next iterations until a pre-determined threshold is reached. Following the stop-and-copy, the new Container at the destination host resumes and the Container at the source host is suspended. The writing of data is one of the key problems with pre-copy. The amount of data written is directly related to the amount of downtime and migration time. More data written equals more dirty pages, and more dirty pages equals more pushes and iterations, which lengthens the migration process. However, it is effective in reading data. Compared to cold migration, it requires less downtime [2].

Post-Copy: It is based on the pull phase and stop-and-copy. A very small portion of the Container is initially transferred from the source to the destination host while the Container at the source host is initially suspended. The Container is then instantly restarted at the destination host. The Container's performance suffers during this period since not all of the memory pages have been moved. Following that, the pull phase starts. If a certain page hasn't been transferred yet, the destination sends a direct on-demand pull request to the source site, which retrieves the necessary pages. It is known as a page fault. Reading data is one of the major post-copy challenges. Page faults increase as read requests increase. This results in more pull requests, which lengthens the time it takes to migrate. Additionally, no service can be offered if the destination is down by accident. In comparison to pre-copy migration, the migration process takes less time [29].

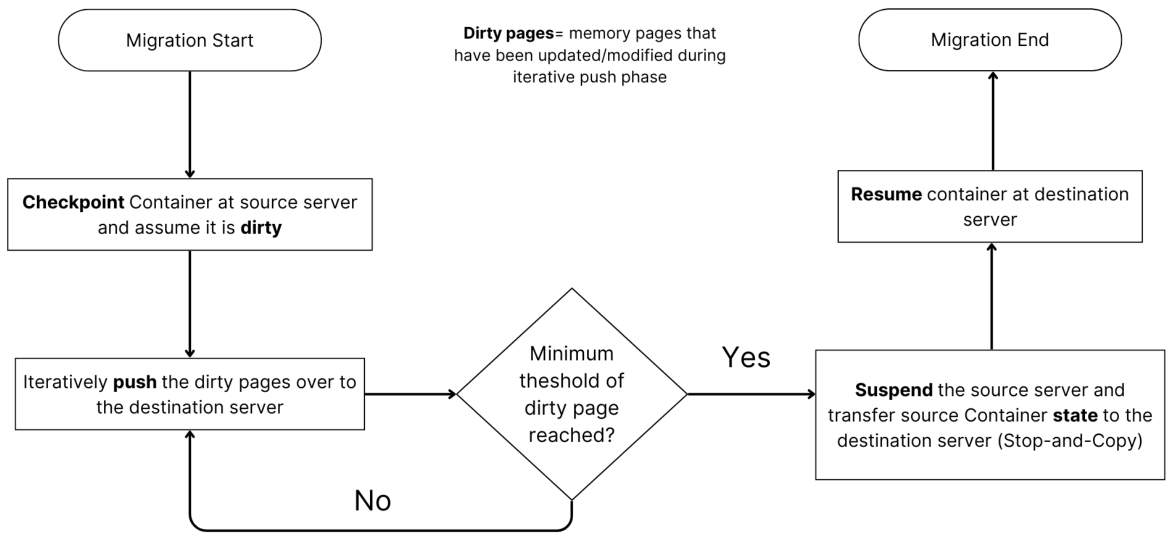
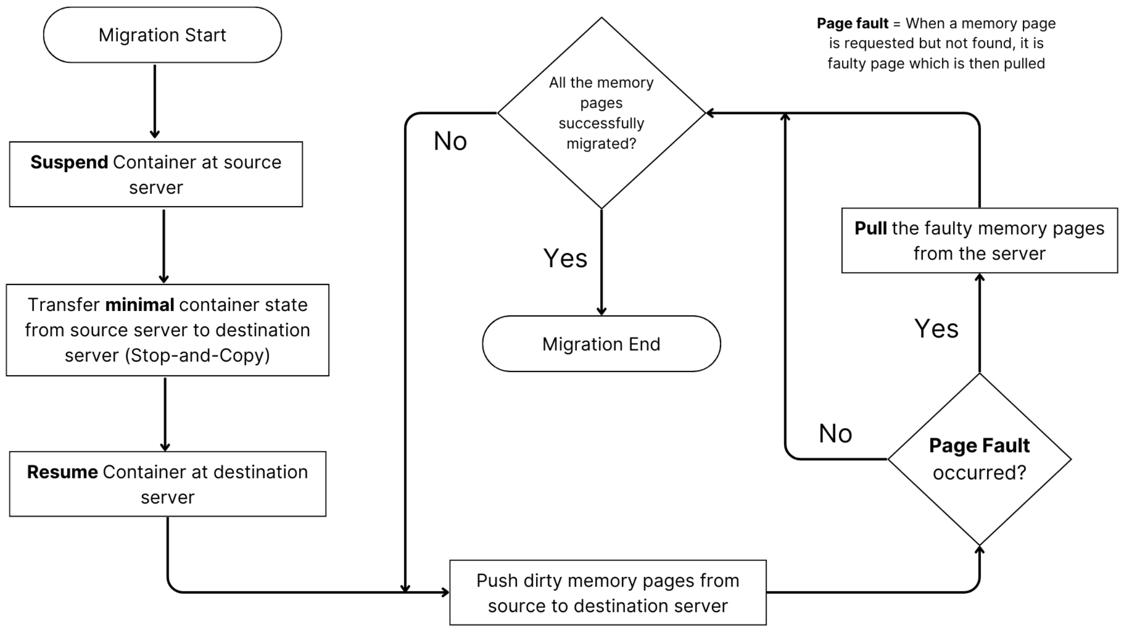
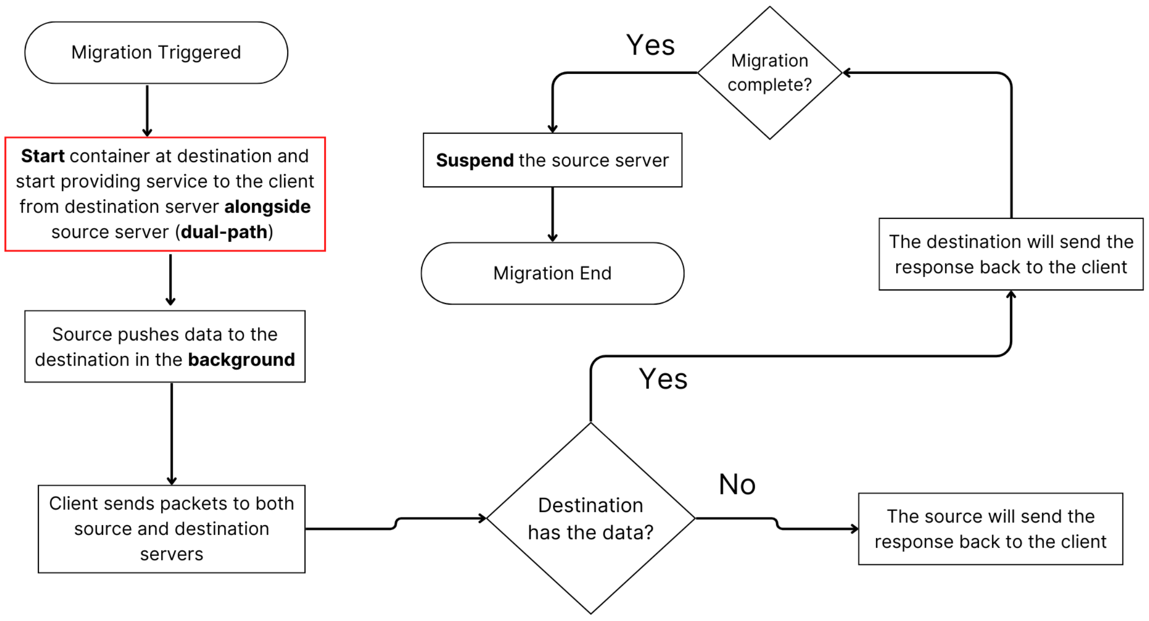

Migration Start
Dirty pages= memory pages that have been updated/modified during iterative push phase
Migration End
Checkpoint Container at source server and assume it is **dirty**
Resume container at destination server
Iteratively **push** the dirty pages over to the destination server
Minimum theshold of dirty page reached?
Yes
No
Suspend the source server and transfer source Container **state** to the destination server (Stop-and-Copy)

Figure 2.19: Pre-copy migration



Migration Start
Suspend Container at source server
Transfer minimal container state from source server to destination server (Stop-and-Copy)
Resume Container at destination server
Push dirty memory pages from source to destination server
Page Fault occurred?
Yes
Pull the faulty memory pages from the server
No
All the memory pages successfully migrated?
No
Yes
Migration End
Page fault = When a memory page is requested but not found, it is faulty page which is then pulled

Figure 2.20: Post-copy migration



Migration Triggered
Start container at destination and start providing service to the client from destination server alongside source server (dual-path)
Source pushes data to the destination in the background
Client sends packets to both source and destination servers
Destination has the data?
No
The source will send the response back to the client
Yes
The destination will send the response back to the client
Migration complete?
Yes
Suspend the source server
Migration End

Figure 2.21: Dual-Path Migration

Advantages of dual-path migration

- Since QUIC is implemented in user-space updating it does not require kernel changes
- The user will not face any service downtime
- The user will not have to suffer performance degradation insha Allah
- There is no need to change any existing infrastructure
- It is easier for servers having multiple clients to migrate in this fashion.

Disadvantages of dual-path migration

- If the old server crashes before completing data migration, then the new server will not be able to provide full service.
- When the old server handles any request, the updates to the state (deltas) need to be pushed to the new server.
- There needs to be an application-specific synchronisation mechanism between the servers during migration.

The transport layer will have two connection modules, each module will be connected to a server. A copy of the same packet will be sent to each server, so the output buffer will create a copy of the payload and send it to each connection module.

If the new server has the data to respond to a client request it will do, otherwise the old server will respond. Both servers know how much data has been transferred so they can deterministically determine which server will send.

If the client uploads some data, then if the new server has already received the file to be written to then it will update it. Otherwise, the old server will update the file and send the updated file to the new server [30].

The priority of each file will be set by the application, files with higher priority will be migrated first to the new server. The synchronisation algorithm will be handled by the application layer, according to its needs and specifications.

Disadvantages of downtime and page faults

- Downtime results in the client being unable to use the service.
- Removing downtime will mean the client will be connected to the server during the whole process of migration
- Page faults allow the client to use the service but the QoS deteriorates, as the destination server has to retrieve the data from the source server first before sending it to the client.
- Getting rid of downtime and page faults will result in seamless migration, where the client will not experience any changes

Chapter 3

Proposed Method

Cloud services are provided via containers that are processes running inside of the servers. Due to a number of conditions such as load balancing, resource balancing, hardware failure or maintenance etc. the container has to be migrated from one server to another. Traditional live migration techniques like Pre-Copy and Post-Copy consists of three rudimentary phases: iterative push phase, pulling of faulted pages and stop-and-copy (control transfer). During the control transfer phase the cloud service is unavailable and suspended until the container state is fully replicated to another target server. This introduces a downtime hampering the end user's quality of experience. Also, pulling faulted pages involves performance degradation which is not desirable. To mitigate the limitations identified in the traditional live migration techniques, we formulate the dual-path migration scheme.

3.1 Dual Path Server Migration with QUIC

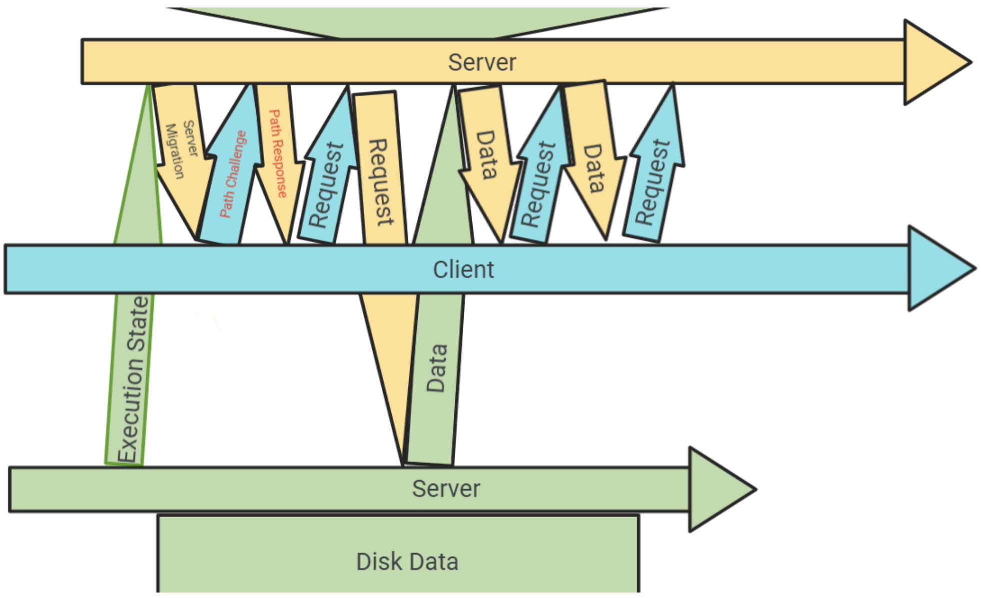
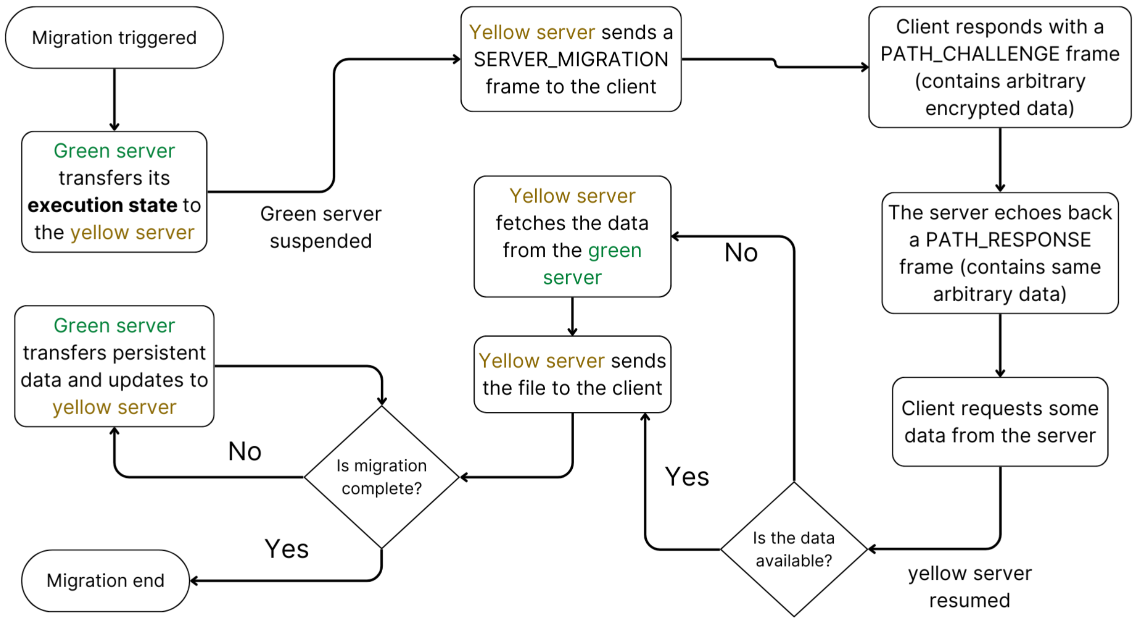


Figure 3.1: Post-Copy Single Server-side Migration in QUIC

Let us now consider a green server serving a turquoise client, the server has to migrate to a new machine which has a different IP address. The steps of the mechanisms are as follows:

1. The green server will transfer it's execution state (CPU, Registers, QUIC Connection Information such as Cryptographic keys to the yellow server)
2. Afterwards, the green server will continue to transfer all of its persistent data and any updates to its state to the yellow server until migration is complete.
3. The yellow server sends a SERVER_MIGRATION frame
4. The client responds with a PATH_CHALLENGE frame which contains encrypted arbitrary data.
5. The server echos backs the same arbitrary data in a PATH_RESPONSE frame.
6. The client requests some data from the server.
7. The server has not yet received this file, so it sends a request to the green server.
8. The green server sends the file.
9. The yellow server can now send the data from the file.
10. The client sends another request.
11. Since the server has the file, it can directly send the data.
12. Once data migration has been completed, all further requests are handled by the yellow server independently.



Migration triggered
Green server transfers its **execution state** to the yellow server
Green server suspended
Yellow server sends a SERVER_MIGRATION frame to the client
Client responds with a PATH_CHALLENGE frame (contains arbitrary encrypted data)
The server echoes back a PATH_RESPONSE frame (contains same arbitrary data)
Client requests some data from the server
yellow server resumed
Is the data available?
Yes
No
Yellow server fetches the data from the green server
Yellow server sends the file to the client
Green server transfers persistent data and updates to yellow server
Is migration complete?
No
Yes
Migration end

Figure 3.2: Step-by-step Post-Copy Single Server-side Migration in QUIC

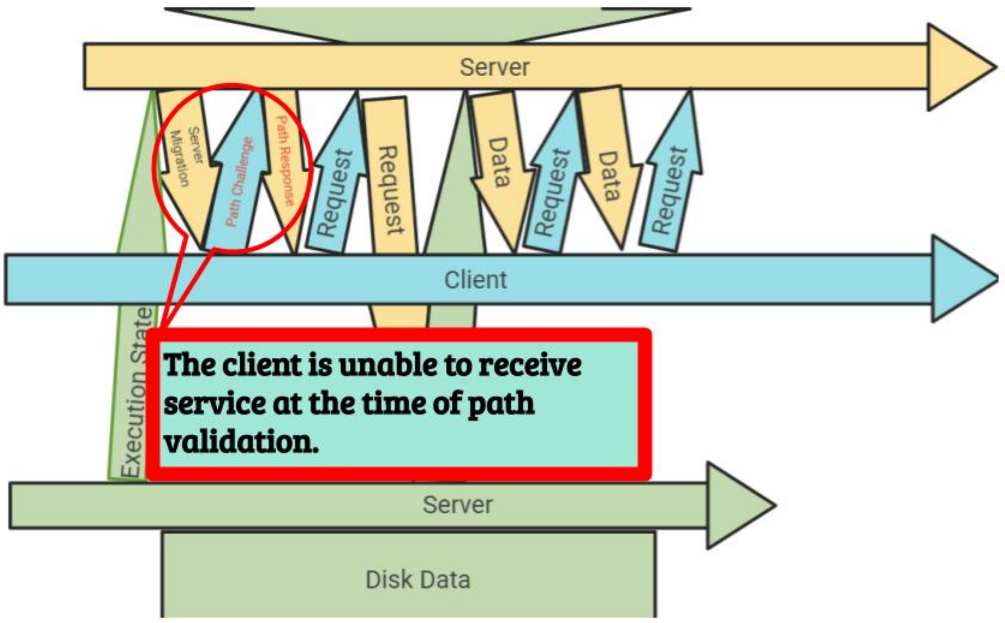
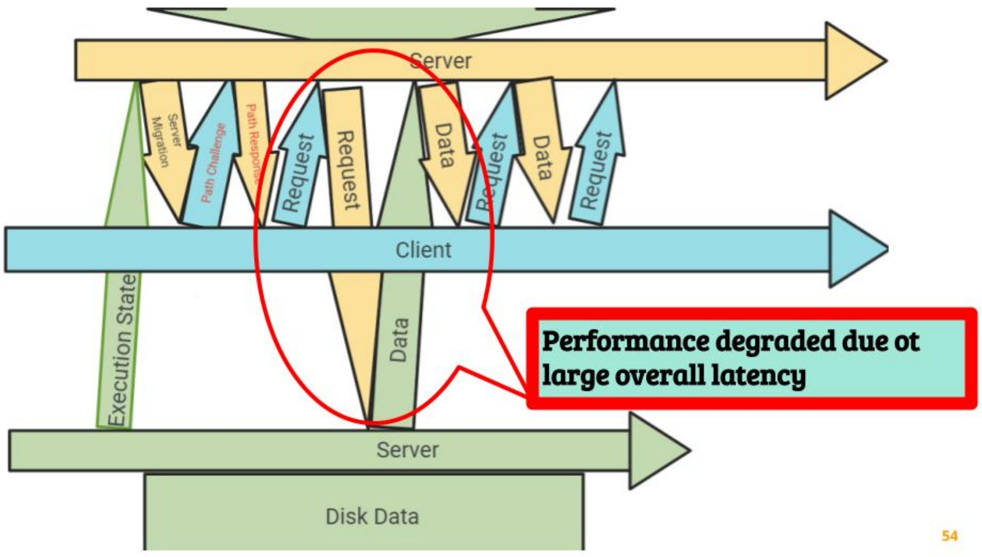


Figure 3.3: Problem of Single Path Server Migration

However, we immediately observe that the client is actually having to wait at least an RTT, to validate the server migration. Hence, it cannot actually get any data from the server until it completes the validation.



Not only that but the post-copy mechanism has the problem of large performance degradation due to page faults occurring at the yellow server, which requires it to pull the required data from the previous green server costing it extra latency for the client.

In order to solve the problem illustrated beforehand, we propose a new mechanism for migration, in this scheme the client will not release connectivity to the green server until after all the disk data has been migrated to the yellow server. This means the client will actually be able to send requests to both servers simultaneously and if the yellow server has the required data to process the request it will send the data, otherwise the green server will respond to this request. However, this dual connection will only be active during the migration phase and the client will return to single path communication once the migration has been completed.

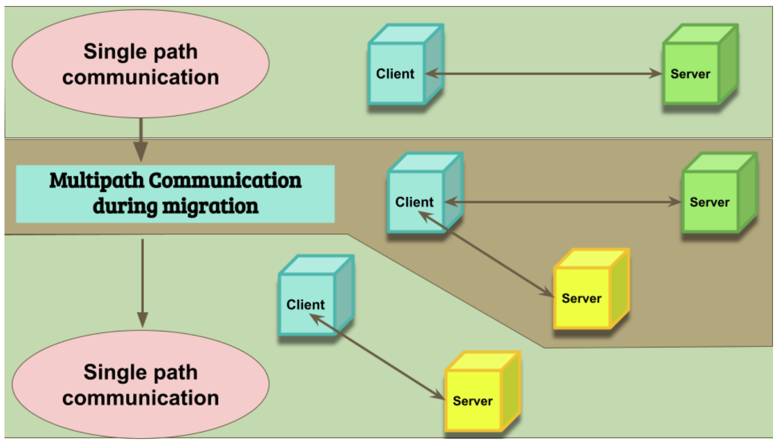

Single path communication
Client
Server
Multipath Communication during migration
Client
Server
Server
Client
Server
Single path communication

Figure 3.4: The main idea behind multipath migration scheme

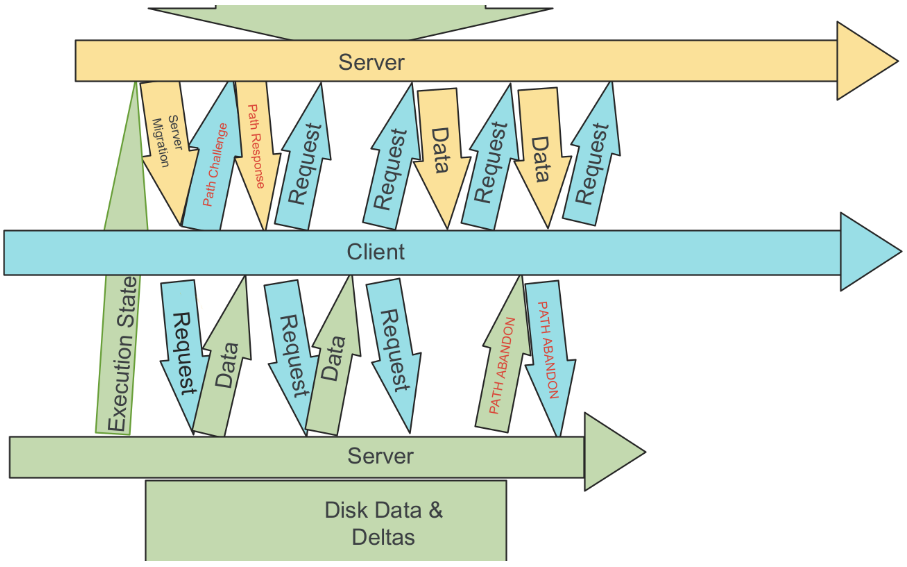


Figure 3.5: Proposed Multi-path Server Migration

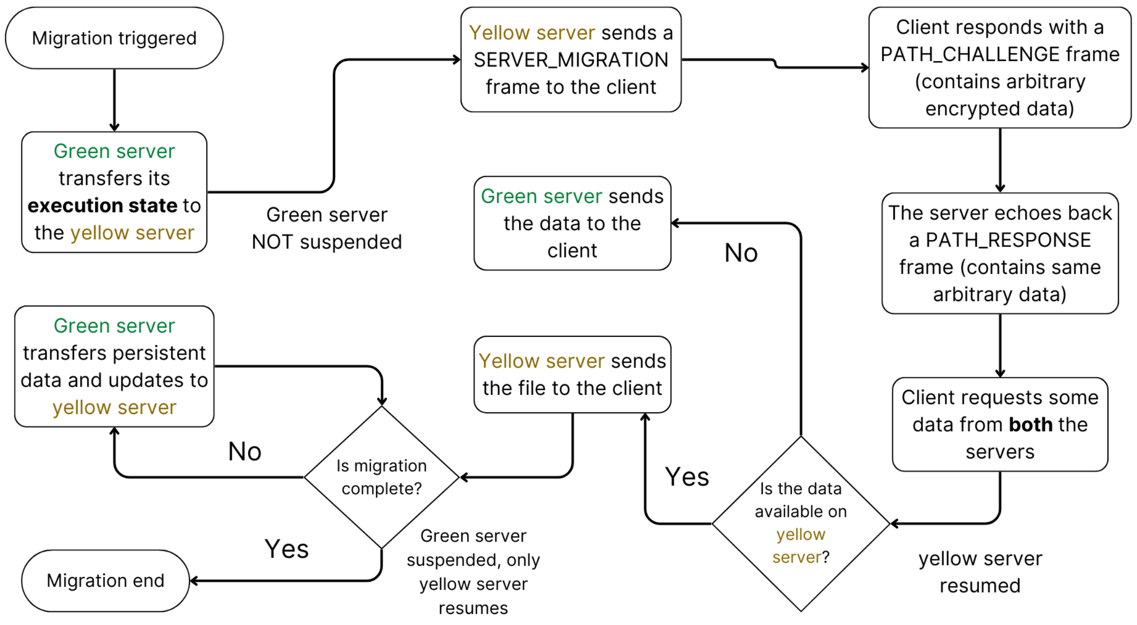

Migration triggered
Green server transfers its execution state to the yellow server
Green server NOT suspended
Yellow server sends a SERVER_MIGRATION frame to the client
Client responds with a PATH_CHALLENGE frame (contains arbitrary encrypted data)
The server echoes back a PATH_RESPONSE frame (contains same arbitrary data)
Client requests some data from both the servers
yellow server resumed
Is the data available on yellow server?
No
Green server sends the data to the client
Yes
Yellow server sends the file to the client
Is migration complete?
No
Green server transfers persistent data and updates to yellow server
Yes
Migration end
Green server suspended, only yellow server resumes

Figure 3.6: Step-by-step Proposed Dual-path Server Migration

Let us now have a walkthrough of the new solution in sha Allah:

1. We have a client and two servers. The client is initially connected to the green server and will eventually migrate to the yellow server.
2. Firstly, the green server will transfer it's execution state, e.g:
   - CPU data
   - Registers
   - QUIC Connection Information
3. Then, the yellow server will send a SERVER_MIGRATION frame to the client
4. Meanwhile the client can continue to receive service from the old server without disruption.
5. The client tries to validate the new server by sending a PATH_CHALLENGE FRAME
6. Meanwhile the green server responds by sending the requested data.
7. PATH_RESPONSE frame is returned by the yellow server, Now the client is simultaneously connected to both servers !!
8. The client will echo the same request to both servers !!
9. Here, since the yellow server is yet to receive the file,the green server responds to this request.
10. The client again sends another request to both servers.

11. Old server is aware via Acknowledgements that the requested data has already been sent to the new server, so it refrains and the yellow server responds to this particular request.

12. The data migration is complete, and the green server can now be released. So, as the client continues communicating with the yellow server and the green server informs the client that it has to stop communicating with it by sending the PATH_ABANDON frame and the client responds likewise.

Please note that if the client uploads any data to the server, then it will send the same datum to both servers, but the old server will update its files and will then push those to the new server along with the other migrating data as deltas. The new server upon receiving this deltas will update its own files. This ensures that the data on both servers are synchronized throughout duration of the migration.

3.2 Implementation of Live Migration Phases with Docker

In the original paper [25], the researcher had utilized `runC` to spawn containers, `CRIU` for checkpoint/restore functionalities and `rsync` to sync files between remote/local servers to demonstrate different migration techniques. We could successfully configure `rsync`. We used `Docker` to replicate spawning containers and checkpoint/restore functionalities. We demonstrated different migration techniques to some extent, we were unable to implement the dual-path migration. Following is the details of our endeavor:

3.2.1 Required Tools

- Oracle VM VirtualBox 6.1.44
- Ubuntu 20.04.4 Focal Fossa 64-bit VDI
- Docker for Ubuntu
- OpenSSH-Server
- Rsync

3.2.2 Container Migration Environment

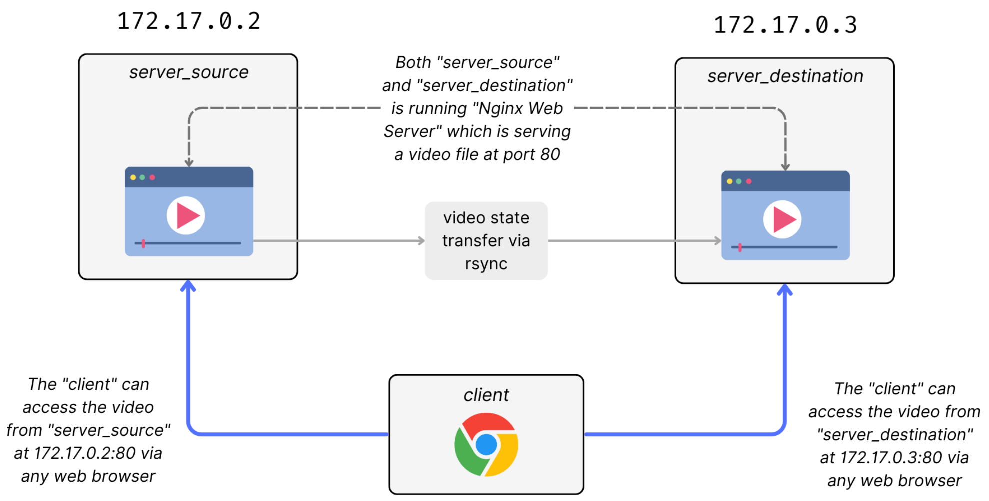

172.17.0.2
172.17.0.3
server_source
server_destination
Both "server_source" and "server_destination" is running "Nginx Web Server" which is serving a video file at port 80
video state transfer via rsync
client
The "client" can access the video from "server_source" at 172.17.0.2:80 via any web browser
The "client" can access the video from "server_destination" at 172.17.0.3:80 via any web browser

Figure 3.7: Nginx Web Server container/process state migration between two servers in the same network

We have to set up two servers (`server_source` and `server_destination`) within the same network as shown in the diagram. Each of the servers should be running Nginx Web Server that will be serving any file type (e.g. video/mp4). In the same network, a client should be able to access the files from any of the two servers. In our case, the client is closer to the `server_source`. So, at first, the video file will be served to the client from `server_source`. At a certain timeframe, when the client will move closer to `server_destination`, we demonstrate Container/Process Migration by:

- **Stop-and-Copy**: Save the video state at `server_source` and stop serving the video from `server_source`. Then, transfer the video state to `server_destination`. Then, read the video state and resume serving the video to the client from `server_destination` at the exact timestamp as it was stopped at `server_source`.
- **Push phase**: Push the required data to `server_destination` from `server_source`.
- **Pull phase**: Pull the required data from `server_source` to `server_destination` on demand.

3.2.3 Environment Setup

First, we have to set up a Linux VM via Hypervisor on Windows machine. So, we should install Oracle VM VirtualBox 6.1.44 and mount pre-installed Ubuntu 20.04.4 Focal Fossa 64-bit VDI (Virtual Drive Image) with 4 CPUs and 8GB RAM. We also have to install Docker on the Ubuntu 20.04 VM. Note that, this Ubuntu VM will now be considered as the “host” machine, although it itself is actually a guest machine running on top of physical Windows. On the host machine (Ubuntu):

```
$ sudo apt update
$ sudo apt install docker.io
```

Before proceeding, we have to enable the root privilege via `sudo -i` to avoid having to write “sudo” every time. We have to create two docker containers that will act as two standalone servers (`server_source` and `server_destination`). The servers should be on the same network so that they are discoverable to each other. Docker, by default, puts the containers in `172.17.0.0/16` (bridge network). The containers will be spawned in this network as two isolated servers. We will use the ubuntu image to create the containers:

```
$ docker run -dit --name server_source -p 8080:80 ubuntu
$ docker run -dit --name server_destination -p 9090:80 ubuntu
```

To view the status of the freshly created servers we issue the `docker ps -a` command which outputs the following:

```
CONTAINER ID ... STATUS   ...   NAMES
5c7c935510b1 ... Up 4 seconds ... server_destination
155e373b5ce1 ... Up 27 seconds ... server_source
```

To view the IP addresses of the servers assigned by docker, we use the `docker inspect` command on both the servers:

```
$ docker inspect server_source | grep IPAddress
"IPAddress": "172.17.0.2"
$ docker inspect server_destination | grep IPAddress
"IPAddress": "172.17.0.3"
```

Now, we have to login to the servers (`server_source` at first, then `server_destination`) and install the necessary dependencies and configure them accordingly so that they communicate with each other. Let us start with the `server_source` at first:

```
$ docker exec -it server_source /bin/bash
```

Before proceeding, on `server_source`, we need to install some tools like ”iputils-ping” to check discoverability of other hosts on the network, ”nano” for editing files, ”openssh-server” for secure remote connection, ”rsync” for transferring files and ”criu” for checkpoint/restore functionalities. So, let’s update the package manager and install the necessary dependencies:

```
$ apt update && apt install -y iputils-ping nano openssh-server rsync criu
```

Similar to `server_source`, we must login to `server_destination` via `docker exec` and install the necessary dependencies there as well:

```
$ docker exec -it server_destination /bin/bash
$ apt update && apt install -y iputils-ping nano openssh-server rsync criu
```

To test discoverability, we can ping `172.17.0.3` (`server_destination`) from `172.17.0.2` (`server_source`) and vice versa. Now, to ensure that `server_source` can establish a secure connection with server_destination, we have to create an SSH key on server_source and copy that key over to server_destination. To generate SSH key, on the server_source:

```
$ ssh-keygen -t rsa
$ cat ~/.ssh/id_rsa.pub
```

We have to copy this key and paste it on server_destination:

```
$ mkdir -p ~/.ssh
$ touch ~/.ssh/authorized_keys
$ chmod 600 ~/.ssh/authorized_keys
$ cat > ~/.ssh/authorized_keys
```

On both servers issue **service ssh restart** to restart SSH. Now, both servers can securely communicate with each other. When creating the servers via docker run command we specified port numbers. For example, for server_source, we specified port 8080:80 which means that any request made to port 8080 on the host machine (127.0.0.1:8080) will be forwarded to port 80 inside server_source (172.17.0.2:80). Similarly, port 9090 on the host machine (127.0.0.1:9090) is mapped to port 80 on the server_destination (172.17.0.3:80). Now, our goal is to create and serve two Nginx Web Servers (one on server_source port 80 and the other on server_destination port 80). To do so, on both servers, let's issue **apt install nginx** to install the latest Nginx Web Server. To start the Nginx web server, on both server_source and server_destination, we issue **service nginx start**. Here, we have to remember, the host machine (Ubuntu), is the client. So, from any browser (Chromium-based browser recommended) on the Ubuntu host machine, hitting 172.17.0.2 or 172.17.0.3 will show a welcome message.

Let us configure the Nginx web server default behavior to serve a video. On both server_source and server_destination,

```
$ cd /var/www/html
$ wget http://commondatastorage.googleapis.com/gtv-videos-bucket/sample/
ForBiggerEscapes.mp4 -O video.mp4
$ touch index.html
$ touch vState.json
```

Here, on both servers, we create an index.html that will serve the video.mp4 by reading the state/timestamp of the video from vState.json file which contains the time records of the video as a JSON object. Initially it's set to {"hours": 0, "minutes": 0, "seconds": 0}. And the contents of index.html will be:

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Nginx Web Server</title>
    <script>
      window.addEventListener("DOMContentLoaded", (w) => {
        const xhr = new XMLHttpRequest();
        xhr.overrideMimeType("application/json");
        xhr.open("GET", "vState.json", true);
        xhr.onreadystatechange = function () {
          if (xhr.readyState === 4 && xhr.status === 200) {
            const vState = JSON.parse(xhr.responseText);
            const video = document.getElementById("video");
            video.currentTime = vState.seconds;
            video.play();
          }
        };
        xhr.send();
      });
    </script>
    <style>
      body {padding:0;margin:0;}
      div {
        display:flex;
        width:100vw;height:100vh;
        background-color:#c2c2c2;
      }
      video {
        top:50%;left:50%;
        width:80%;position:absolute;
        transform:translate(-50%,-50%);
      }
    </style>
  </head>
  <body>
    <div>
      <video id="video" src="video.mp4" controls></video>
    </div>
  </body>
</html>
```

Now, on both servers, we issue `service nginx reload` to reload the server configurations.

3.2.4 Observation on Stop-and-Copy Phase

At first, we get the video file will be served from `server_source` to the client. On Ubuntu host machine, we go to `172.17.0.2` via the Chromium browser with the necessary development flags:

```
$ chromium --disable-web-security 172.17.0.2
```

The client can see a video running from the very first frame. Now, we play the video until `4:50` and pause it. At this point, we have to save the video state on `server_source`. So, the `vState.json` will contain `{"hours": 0, "minutes": 4.833, "seconds": 290}`. We have to transfer this file remotely over to `server_destination` via `rsync`. On `server_source`:

```
$ rsync -avzP /var/www/html/vState.json root@172.17.0.3:/var/www/html
sending incremental file list
vState.json
            60 100%    0.00kB/s    0:00:00 (xfr#1, to-chk=0/1)

sent 167 bytes received 41 bytes 416.00 bytes/sec
total size is 60 speedup is 0.29
```

The `vState.json` file at `server_destination` is now synced with *server_source*. We have to issue the `service nginx reload` command to reload the server configurations. At this point, the video is ready to be resumed from `server_destination` to the client at the exact timestamp where it was paused on `server_source`. So, on Ubuntu host machine, we go to `172.17.0.3` via the Chromium browser with the necessary development flags:

```
$ chromium --disable-web-security 172.17.0.3
```

The video is now served from `server_destination` which is resumed at `4:50`. This entire process demonstrates the *Stop-and-Copy* phase of migration.

3.2.5 Observation on Push phase

In observation 1, we only migrated the state which is less than 100 bytes in size. But, we know in Pre-Copy or Hybrid Migration there is a push phase. In push phase, before even migrating the state from `server_source` to `server_destination` we have to transfer the actual data or memory pages. This data is often very large in size from a few Megabytes to many Gigabytes. In this section we use `rsync` to migrate large files from `server_source` to `server_destination`. To create sample files with specific file sizes (e.g. 1GB = 1,073,741,824 Bytes), we used the following python script:

```
with open("file.bin", mode='wb') as f:
    f.truncate(1073741824)
```

After that, issued `rsync` with appropriate flags like `-a` to enable archive mode that preserves permissions, ownership, timestamps, etc, `-v` to get detailed output during the synchronization process, `-P` to show the progress of the transfer and to resume partially transferred files if the transfer is interrupted. Then we monitored the output:

```
$ rsync -avP --stats /var/www/html/file.bin root@172.17.0.3:/var/www/html
sending incremental file list
file.bin
  1,073,741,824 100% 522.97MB/s  0:00:01 (xfr#1, to-chk=0/1)
```

```
Number of files: 1 (reg: 1)
Number of created files: 1 (reg: 1)
Number of deleted files: 0
Number of regular files transferred: 1
Total file size: 1,073,741,824 bytes
Total transferred file size: 1,073,741,824 bytes
Literal data: 1,073,741,824 bytes
Matched data: 0 bytes
File list size: 0
File list generation time: 0.001 seconds
File list transfer time: 0.000 seconds
Total bytes sent: 1,074,004,067
Total bytes received: 35

sent 1,074,004,067 bytes received 35 bytes 429,601,640.80 bytes/sec
total size is 1,073,741,824 speedup is 1.00
```

From the above information, `file.bin` is a file of size 1GB (1,074,004,067 Bytes). Rate of transfer is 522.97MB/s which takes 1.95s to complete the transfer. 262KB of additional data was needed to be sent and 35 Bytes of data was received for synchronization purposes. `rsync` provides `-z` flag to compress the data being transferred to save bandwidth. Using compression we get:

```
$ rsync -avzP --stats /var/www/html/file.bin root@172.17.0.3:/var/www/html
sending incremental file list
file.bin
  1,073,741,824 100%  2.57GB/s   0:00:00 (xfr#1, to-chk=0/1)

Number of files: 1 (reg: 1)
Number of created files: 1 (reg: 1)
Number of deleted files: 0
Number of regular files transferred: 1
Total file size: 1,073,741,824 bytes
Total transferred file size: 1,073,741,824 bytes
Literal data: 1,073,741,824 bytes
Matched data: 0 bytes
File list size: 0
File list generation time: 0.001 seconds
File list transfer time: 0.000 seconds
Total bytes sent: 32,889
Total bytes received: 35

sent 32,889 bytes received 35 bytes 21,949.33 bytes/sec
total size is 1,073,741,824 speedup is 32,612.74
```

In this case, rate of transfer is 2.57GB/s which takes 0.4s to complete the transfer. 32,889 Bytes of additional data was needed to be sent and 35 Bytes of data was received for synchronization purposes.

3.2.6 Collected Data

Table 3.1: Collected data for remotely “pushing” original and compressed data via `rsync` to `172.17.0.3` from `172.17.0.2`

File size	**Without compression**			**With compression**		
	Transfer rate (MB/s)	Speedup (:1)	Total time (s)	Transfer rate (GB/s)	Speedup (:1)	Total time (s)
100MB	354.50	1.00	0.28	2.57	31,291.44	0.038
500MB	542.27	1.00	0.9	3.70	32,463.65	0.132
1GB	522.97	1.00	1.95	2.57	32,612.74	0.4
5GB	387.82	1.00	13.2	1.96	32,731.44	2.55
10GB	448.22	1.00	22.4	1.85	32,746.41	5.4

3.2.7 Mvfst

Mvfst is an open source implementation of QUIC developed by Meta. It is written in C++17 and was planned to be used to implement dualpath extension by modifying the source code. However, it was not possible to realise this plan and the new frames were added but all other progress could not be made. The main was the lack of detailed documentation on the structure and inner workings of the codebase.

3.2.8 Proxygen

Proxygen contains the core C++ HTTP abstractions. It is developed by Meta. It is used as the basis for building many HTTP servers, proxies, and clients. It also supports HTTP/3 and Quic, making it suitable for the project. The goal was to use proxygen to build two servers and one client. It was however not possible due to errors during building. The main error that persisted was an internal compiler error. No amount of troubleshooting could solve this problem.

3.3 Mathematical Analysis

3.3.1 Model Definition

In this section we develop two models post-copy and dual-path and mathematically analyse them using probability theory to determine which model is expected to perform better and under what circumstances. We make some assumptions to simplify our model to make it mathematically comprehensible, namely:

1. The data is being migrated from the old server to the new server at a constant uniform rate.

2. The round trip times (RTT) between the three entities (2 servers and 1 client) remain unchanged during migration.

3. If there is a need for synchronization between two servers then it is done instantly so it has negligible effect.

4. The amount of data requested by the client is constant.
5. The data requested by the client is independent of the previous requests it has made.
6. The data requested by the client is random i.e. it is equally likely that any data on the server can be requested by the client
7. There are no packet losses and there is no packet fragmentation at the IP layer.
8. The time it takes for processing the packet is negligible compared to the RTT

Table 3.2: List of parameters considered in the model

Variable	**Restriction**	**Description**
t	$0 \leq t \leq t'$	Time elapsed since server migration frame received
s		The round-trip time from new server to old server
t'	$t' > t$	The time duration to complete the migration
r_{new}		The round-trip time between new server and client
r_{old}		The round-trip time between old server and client
Client Response Time (CRT)		The time it takes for a client to receive the response of a request to a server

3.3.2 Calculating the probability that the requested data is found at the new server

Since, data is being migrated to the new server at a constant rate, the probability of a certain piece of data being present in the new server should increase uniformly with time.
Let us define two events R and $\sim R$ (because they are complementary events). R is the case when the client requests a certain piece of data and it is found in the new server $\sim R$ is the case when the client requests a certain piece of data and it is **not** found in the new server
$P(R) = t/t'$
Hence, $P(\sim R) = 1 - P(R) = 1 - (t/t')$

3.3.3 Calculating the expected client response time conditioned upon time

E is the expected value, our objective here is to find conditional expected CRT given the conditional probabilities and their outcomes in terms of the RTT. The expected outcome is the weighted sum of costs where each cost is weighted by the likelihood of that cost. In our case the cost of each event is the CRT in case of that particular event. $E[Client\ Response\ Time] = P(R) * CRT[R] + P(\sim R) * CRT[\sim R]$

In the case of dual-path, in the case of event R, the new server will respond with the data so the CRT will be r_{new}, but if it does not occur then, the data will be fetched from the old server so CRT will be r_{old}. In the case of the post-copy scheme, if R occurs then similarly to dual-path the new server will respond. But if the event $\sim R$ occurs then, the news server will have a page fault and will have to obtain the data from the old server (requiring s time) and then send it to the client, so the CRT will be $s + r_{new}$.

Table 3.3: Table showing CRTs in various scenarios

Client Response Time (CRT)	**Event**	
Scheme	R	$\sim R$
Dual path	r_{new}	r_{old}
Post copy	r_{new}	$r_{new} + s$

We have to split our migration timeline into two parts, the first part is during path validation, i.e. when PATH_CHALLENGE and PATH_RESPONSE frames are being exchanged. In both migration schemes, it requires r_{new} time. However, in the background during this time data is being migrated from the old server to the new server. So, we consider two time periods , $0 \leq t \leq r_{new}$ and $r_{new} < t \leq t'$

3.3.4 Dual Path

For dualpath in the time period $0 \leq t < r_{new}$, the client is prohibited to exchange data with the new server until the PATH RESPONSE frame is received. So, the client will only send requests to the old server and it will take r_{old} time in sha Allah. $E_{dualpath}[Client\ Response\ Time] = r_{old}$

For dual path in the time period $r_{new} < t \leq t'$,

$$\begin{aligned} E_{dualpath}[Client\ Response\ Time] &= P(R) * CRT[R] + P(\sim R) * CRT[\sim R] \\ &= P(R)r_{new} + P(\sim R)r_{old} \\ &= t/t' * r_{new} + (1 - t/t')r_{old} \\ &= t/t'(r_{new} - r_{old}) + r_{old} \end{aligned}$$

3.3.5 Postcopy

For postcopy in the time period $0 <= t < r_{new}$, the client is prohibited to exchange data with the new server until the PATH RESPONSE frame is received. So, the client cannot send requests to the old server as it has completely disconnected with that server and the CRT in this time period is infinite, (basically no messages are sent)

For post copy in the time period $r_{new} < t \leq t'$,

$$\begin{aligned} E_{postcopy}[Client\ Response\ Time] &= P(R) * CRT[R] + P(\sim R) * CRT[\sim R] \\ &= P(R) * r_{new} + P(\sim R) * (r_{new} + s) \\ &= t/t' * r_{new} + (1 - t/t')(r_{new} + s) \\ &= (1 - t/t')s + r_{new} \end{aligned}$$

3.3.6 Inter-scheme Comparison

Let us now compare and find the scenario when dual path will be superior to post-copy i.e. the response time of post-copy will be higher than that of dual path. In the time period, $0 < t < r_{new}$, dual path already has a lower CRT as its CRT is finite but that of post-copy is infinite. Now, let us mathematically compare in the other time period,

$$E_{dualpath}[Client\ Response\ Time] < E_{postcopy}[Client\ Response\ Time]$$

or, $P(R)r_{new} + P(\sim R)r_{old} < P(R) * r_{new} + P(\sim R) * (r_{new} + s)$
or, $P(\sim R)r_{old} < P(\sim R) * (r_{new} + s)$ [subtracting $P(R)r_{new}$ from both sides]
or, $r_{old} < r_{new} + s$ [Dividing both sides by $P(\sim R)$, it is assumed that $P(\sim R)$ will never be equal to zero in this time frame]
So, when $r_{old} < r_{new} + s$ then dual path will have a lower client response time in sha Allah. Below are two graphs, demonstrating this fact

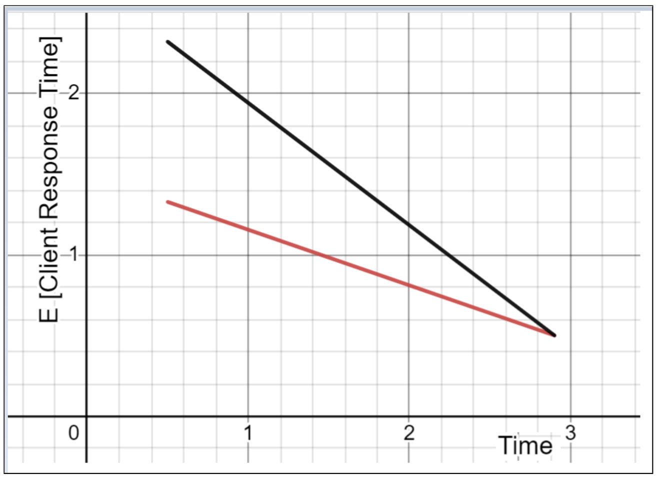


Figure 3.8: Comparison between E_{dual} (red) and $E_{postcopy}$ (black), the parameters are $t' = 2.9, r_{new} = 0.5, r_{old} = 1.5, s = 2.2$

We see in the graph above that since $r_{new} + s = 2.7$ which is greater than $r_{old} = 1.5$ so, $r_{old} < r_{new} + s$ is satisfied and dualpath has an overall lower Expected CRT than postcopy.

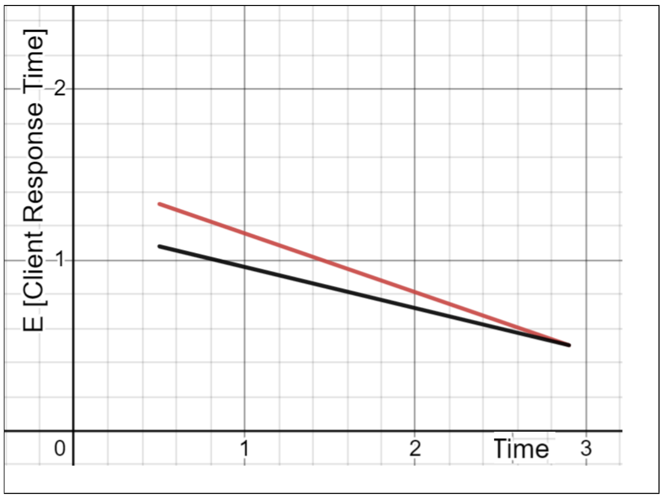


Figure 3.9: Comparison between E_{dual} (red) and $E_{postcopy}$(black), the parameters are $t' = 2.9, r_{new} = 0.5, r_{old} = 1.5, s = 0.7$

We see in the graph above that since $r_{new} + s = 1.2$ which is lesser than $r_{old} = 1.5$ so, $r_{old} < r_{new} + s$ is not satisfied and dualpath has an overall higher Expected CRT than postcopy.

3.3.7 Findings

We can therefore conclude that dualpath is better in the time period $0 < t < r_{new}$, but in the other time period until the end of the migration the superiority of dualpath depends on the on condition $r_{old} < r_{new} + s$ being true.

Chapter 4

Conclusion and Future Works

The goal of our thesis was to work on top of the paper "EXTENDING MVFST TO SUPPORT ENHANCED SERVER-SIDE MIGRATION IN QUIC" [25] via implementing various live container migration techniques like Pre-Copy and Post-Copy. and providing a comparative analysis with dual-path migration using Mvfst and Proxygen. The live container migration techniques depend on three phases Stop-and-Copy, Push phase and Pull phase. In our work, we setup and configured the servers using `Docker` and `Nginx`. We demostrated the migration phases and collected the data via migration tools like `rsync`. However, due to technical difficulties and lack of expertise in this domain (coupled with lack of resources) we as novice students couldn't integrate `Mvfst` or `Proxygen` for dual-path migration for that the future goal of our thesis is to provide a detailed analysis of the comparison between dual-path and live migration schemes in QUIC server side migration.

In the future we aim to implement the dual path protocol detailed in the thesis. MVFST will be used as it contains a vast C++ library of QUIC implementation. Proxygen will be used to set up the servers. Both are made and used by Meta, formerly known as Facebook. CRIU will be used to facilitate the migration process. We will simulate an environment where live migration will take place while the client will receive service from the servers. We aim to compare the performace of the dual path protocol with the other existing protocols using experimental data. We will also see whether there is any measureable performace degradation, or downtime - things that affect quality of service during the migration process. That will help establish the dual path migration method as a viable technique of live migration.

Bibliography

[1] JohnDellaverson, TianxiangLi, YanrongWang, JanaIyengar, A. Afanasyev, and LixiaZhang, "A Quick Look at QUIC," 2018.

[2] P. Kumar, J. Chen, and B. Dezfouli, "Quicsdn: Transitioning from tcp to quic for southbound communication in sdns," 07 2021.

[3] J. Iyengar, E. Fastly, M. Thomson, and E. Mozilla, "RFC 9000 QUIC: A UDP-Based Multiplexed and Secure Transport," 2021.

[4] E. M. M. Thomson and E. s. S. Turner, "RFC 9001 Using TLS to Secure QUIC," 2021.

[5] D. Stenberg, "Why QUIC [https://http3-explained.haxx.se/en/why-quic]," 2018.

[6] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. B. Krasic, C. Shi, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. C. Dorfman, J. Roskind, J. Kulik, P. G. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, and W.-T. Chang, "The quic transport protocol: Design and internet-scale deployment," 2017.

[7] M. Engelbart and J. Ott, "Congestion control for real-time media over quic," in *Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC*, EPIQ '21, (New York, NY, USA), p. 1–7, Association for Computing Machinery, 2021.

[8] P. Kharat, A. Rege, A. Goel, and M. Kulkarni, "Quic protocol performance in wireless networks," pp. 0472–0476, 04 2018.

[9] M. Palmer, T. Krüger, B. Chandrasekaran, and A. Feldmann, "The quic fix for optimal video streaming," 09 2018.

[10] V. Vu and B. Walker, "On the latency of multipath-quic in real-time applications," pp. 1–7, 10 2020.

[11] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport." RFC 9000, May 2021.

[12] T. Völker, E. Volodina, M. Tüxen, and E. Rathgeb, "A quic simulation model for inet and its application to the acknowledgment ratio issue," pp. 737–742, 07 2020.

[13] M. Seufert, R. Schatz, N. Wehner, B. Gardlo, and P. Casas, "Is quic becoming the new tcp? on the potential impact of a new protocol on networked multimedia qoe," pp. 1–6, 06 2019.

[14] Q. Coninck and O. Bonaventure, "Multipathtester: Comparing mptcp and mpquic in mobile environments," pp. 221–226, 06 2019.

[15] M. Quadrini, M. Luglio, F. Zampognaro, C. Roseti, and A. Abdelsalam, "Quic-proxy based architecture for satellite communication to enhance a 5g scenario," 06 2019.

[16] R. J. Saleh Alawaji, "IETF QUIC v1 Design," 2021.

[17] J. Zhang, L. Yang, X. Gao, G. Tang, J. Zhang, and Q. Wang, "Formal analysis of quic handshake protocol using symbolic model checking," *IEEE Access*, vol. 9, pp. 14836–14848, 01 2021.

[18] Y. Liu, Y. Ma, Q. D. Coninck, O. Bonaventure, C. Huitema, and M. Kühlewind, "Multipath Extension for QUIC," Internet-Draft draft-ietf-quic-multipath-03, Internet Engineering Task Force, Oct. 2022. Work in Progress.

[19] J.-M. Chen, S. Chen, X. Wang, L. Lin, L. Wang, and J. Cui, "A virtual machine migration strategy based on the relevance of services against side-channel attacks," *Sec. and Commun. Netw.*, vol. 2021, jan 2021.

[20] B. Bajic, I. Cosic, B. Katalinic, S. Morača, M. Lazarevic, and A. Rikalovic, "Edge computing vs. cloud computing: Challenges and opportunities in industry 4.0," 10 2019.

[21] A. Verma and V. Verma, "Comparative study of cloud computing and edge computing: Three level architecture models and security challenges," vol. 9, 08 2021.

[22] S. Chithra, D. Maheswari, and C. Sethurathinam, "A comparative study on cloud computing and edge computing with its applications," vol. 12, 02 2022.

[23] A. Yadav, L. Garg, and R. Mehra, *Docker Containers Versus Virtual Machine-Based Virtualization: Proceedings of IEMIS 2018, Volume 3*, pp. 141–150. 01 2019.

[24] L. Conforti, A. Virdis, C. Puliafito, and E. Mingozzi, "Extending the quic protocol to support live container migration at the edge," in *2021 IEEE 22nd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pp. 61–70, 2021.

[25] D. CASU, "Extending mvfst to support enhanced server-side migration in QUIC: protocol design and performance evaluation," 2022.

[26] M. Kanagarathinam, S. Singh, S. Jayaseelan, M. Maheshwari, G. Choudhary, and G. Sinha, "Qsocks: 0-rtt proxification design of socks protocol for quic," *IEEE Access*, vol. 8, pp. 1–1, 01 2020.

[27] M. Hall-Andersen, D. Wong, N. Sullivan, and A. Chator, "nquic: Noise-based quic packet protection," pp. 22–28, 12 2018.

[28] M. A. Altahat, A. Agarwal, N. Goel, and J. Kozlowski, "Dynamic hybrid-copy live virtual machine migration: Analysis and comparison," *Procedia Computer Science*, vol. 171, pp. 1459–1468, 2020. Third International Conference on Computing and Network Communications (CoCoNet'19).

[29] L. Basyoni, A. Erbad, M. AlSabah, N. Fetais, A. Mohamed, and M. Guizani, “Quictor: Enhancing tor for real-time communication using quic transport protocol,” *IEEE Access*, vol. PP, pp. 1–1, 02 2021.

[30] A. Kyratzis and P. Cottis, “Quic vs tcp: A performance evaluation over lte with ns-3,” *Communications and Network*, vol. 14, pp. 12–22, 01 2022.