POLITECNICO DI TORINO

POLITECNICO DI TORINO

Master Degree course in Communications and Computer Networks Engineering

Master Degree Thesis

Application of Multipath Protocols in Stateful Migration Scenarios

Supervisors
Prof. Paolo Giaccone
Prof. Carla Fabiana Chiasserini

Candidate
Alessio Ferracini

Academic Year 2022-2023

Acknowledgements

This work was supported partially by the European Commission through Grant No.101095890 (project PREDICT-6G) and partially by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE0000001 - program "RESTART").

Abstract

The edge computing architecture has steadily gained traction in the recent years and is foreseen to become more and more widespread. The main appeal is the possibility of deploying time-critical mobile services on cloud-like resources, reducing computational power requirements on mobile devices, especially for applications like autonomous driving or unmanned aerial vehicles (UAVs). While the advantages are clear, there are still many challenges to be faced in this field. They mainly concern the unreliability of the access segment, the limited resources in the edge points of presence (PoP) and the variance in network conditions (both latency wise and bandwidth wise). At the same time multipath protocols are emerging as a way to exploit the multiple network interfaces of modern devices to provide more reliability and larger bandwidth. In order to reap the latency benefits of the shorter distance from the user it is important to support mobility through migration, which can however create service downtimes that cannot be tolerated by real-time applications. In this work we will explore the idea of using multipath protocols to facilitate the stateful migration process and reduce the total downtime. We present a literature review on the subject and comment on the properties of three analyzed multipath protocols: MPTCP, MPDCCP and (MP)QUIC.

Contents

Chapter 1

Introduction

Cloud computing technology relies heavily on the centralization of computing and data resources, so that these resources can be accessed on-demand by the distributed end users and costs are reduced. Cloud services however are provided by large centralized data-centers that may be located far away from the users. Consequently, a user can experience high connection latency which is not compatible with the requirements of Internet of Things (IoT) applications. A considerable part of the computing tasks generated by these applications, such as virtual reality, augmented reality, and industrial control, require timely and context-aware processing. As a result, processing massive data traffic is a key feature of the future Internet and wireless communication systems. Furthermore, Internet and wireless communication networks increasingly view high data rate and low delivery latency as two key performance indices. It implies that powerful computation devices need to process massive data traffic, and high data rate transmission links are also necessary to transfer the data traffic for the Internet and wireless communication networks, respectively. To improve the data throughput and rapid response of mobile devices or sensors, a small cloud can be connected directly via the wireless communication infrastructure at the network edges (e.g., cellular base station and Wi-Fi access point) to provide services to the mobile users within its coverage. In this paradigm, called Mobile edge Computing (MEC), clients act both as data consumers and data producers (as shown in figure 1.1) meaning that they produce data, offload the processing computation to the edge/cloud are sent the resulting output. MEC has emerged as a key enabling technology for realizing the IoT visions as it allows to deploy mobile devices with relatively limited computational and storage capacity, providing high data processing capabilities and robustness and enabling scalability for time sensitive applications. Within the edge computing paradigm, migration becomes essential to preserve the quality of service benefits that are gained by moving the computing power closer to the user devices. The reduced coverage and processing power of a single edge server can lead to significant performance degradation in the case of mobile user terminals if not managed properly. For example, connected vehicle users usually face similar problems: vehicles may run in mountainous areas, mines, tunnels, etc., which can cause connection interruptions when

entering signal dead zones or passively switching base stations. Frequent connection interruptions and slow connection establishment can lead to poor user experience. Seamless service migration is especially important because it allows to maintain the application's internal state, ensuring service continuity and data integrity.

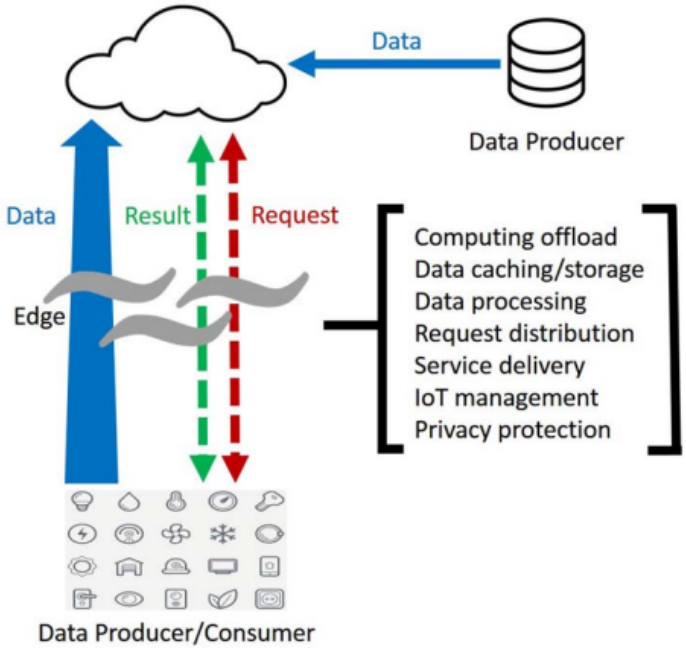


Figure 1.1: Mobile Edge Computing model [1]

As the demand for low-latency and high-throughput applications intensifies, the traditional networking architectures face challenges in meeting these stringent requirements. Multipath protocols represent a promising avenue to address the limitations of conventional communication mechanisms in edge computing environments. By exploiting multiple concurrent paths between source and destination, multipath protocols can mitigate issues related to network congestion, latency, and packet loss. The advent of multipath communication is particularly pertinent in edge computing scenarios where diverse devices generate and consume data at the network periphery. Traditional networking protocols often struggle to maintain optimal performance in these dynamic and heterogeneous environments. Multipath protocols, however, introduce a dynamic approach to data transmission, distributing traffic across several available paths and, consequently, enhancing the overall reliability and efficiency of communication.

1.1 Context and problem statement

Stateful container migration is the process of moving a containerized application from one host or environment to another while preserving the application state and configuration. The biggest challenges to overcome are the limited network bandwidth available in some

edge scenarios and the stringent time constraints on migration duration and downtime, which are especially relevant if the containerized application is handling real-time data [2]. In addition, while migration in the cloud often happens within the same network, this is not the case for edge computing. This is a big downside since now the migration procedure must take into account a change in the IP address of the container. What this implies is that in case of connections that rely on TCP, once the old container is shut down, the connection must be reestablished from the new instance, stretching the total service downtime. We investigate the intuition of using the subflow feature of multipath protocols to avoid having to reestablish this connection. The scenario we are taking into consideration is depicted in figure 1.2: once migration is triggered, the state starts being transferred to the destination; this data includes information on the open connections that can be used by one of the entities involved to open a new subflow between the user and the destination; once migration is completed the old container instance is shut down and data immediately shifts on the backup subflow achieving seamless migration. Many multipath protocols have emerged in the last years, each with its own features.

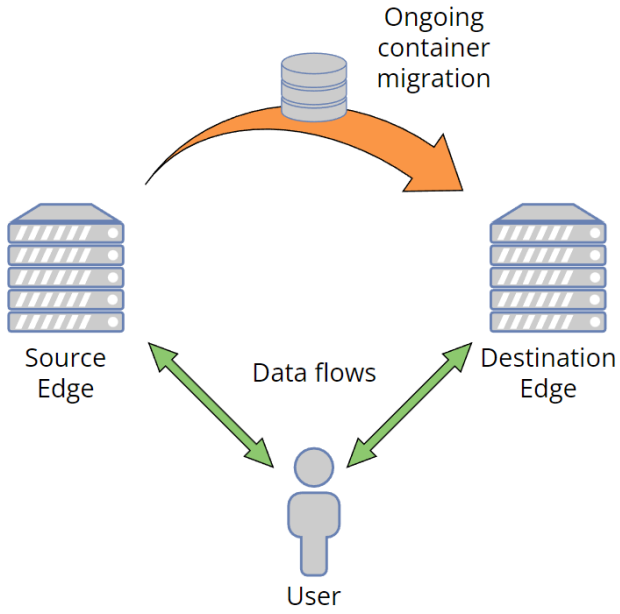


Figure 1.2: Simplified system model

In this paper we present a detailed analysis of the most popular protocols available at the moment, including the usefulness of their features relative to the described scenario, ease of implementation and future prospects.

1.2 Thesis Structure

In this work we will examine the state of the art in the field of stateful container migration, discussing tools and techniques used. After that we will provide a literature review on multipath protocols, discussing the properties of MPTCP, MPDCCP and (MP)QUIC,

with a focus on their application to container migration. We will summarize and discuss a general solution that most of the analyzed papers can be mapped to.

The content is organized as follows:

Chapter 1: Introduction

The first chapter introduces the context and motivation of this thesis. It discusses the importance of seamless migration in the MEC context and how multipath protocols can help achieve it.

Chapter 2: Background Technologies

The second chapter contains a more detailed discussion on the edge computing paradigm and an overview of the technologies involved in container migration, which includes general implementation and features of such tools.

Chapter 3: State of the Art Techniques for Stateful Migration

The third chapter describes more in detail the edge computing paradigm and introduces the state of the art of container migration technologies. We discuss the difference between stateful and stateless migration, following with a state of the art review on techniques for stateful migration.

Chapter 4: Multipath Protocols Applied to Stateful Migration

The fourth chapter homes the bulk of this work. We first provide context on the role and usefulness of multipath protocols in the new 5G networks and document the reasons why they are gaining attention from the research community.

Chapter 5: Conclusions

The fifth chapter recaps the topics presented in this thesis. The advantages and weaknesses of container migration and multipath protocols are discussed.

Chapter 2

Background Technologies

2.1 Containers

Computing virtualization, which consists in a flexible way to share hardware resources between different unmodified operating systems, has steadily gained popularity from the late '90s onward. The main obstacle that it allowed to overcome was the inefficiency caused by the "one application per server" rule: the rule was set because malicious or misbehaving applications should not be able to compromise the integrity of the other services running on the same system, but initially the only solution was to use a separate machine for each service. This resulted in an increasing amount of wasted resources, as power consumption does not grow linearly with CPU utilization. Virtualization offers several advantages that solved this problem. First and foremost it allows different services to run on the same host with an improved degree of isolation, enabling for example the assignment of CPU cores to specific applications. It follows that energy consumption is optimized as the hardware has on average less idle time. The greatest benefit of computing virtualization lies however in its flexibility, also called agility: having complete control over the OS instances running on virtualized hardware allows for example to pause or restart their execution, migrate them to other locations, spawn new instances or reallocate the hardware resources to address peak loads. The main drawback of this technique is the additional overhead added to each application by the OS, however it is considered acceptable for most use cases where hardware is not a major constraint.

Currently there exist two categories of virtualization, each with their own pros and cons: virtual machines and containers. A virtual machine (VM) is a virtual environment that functions as a virtual computer system with its own CPU, memory, network interface, and storage, created on a physical hardware system. It runs on top of a software called Hypervisor or VMM (Virtual Machine Monitor), which treats computing resources as a pool that can be easily relocated. The system on which the hypervisor runs is called host OS, while the many VMs that use its resources are called guests. Operating system-level virtualisation, better known as containerisation, is a virtualisation approach enabled by a

set of operating system features where the kernel itself allows the coexistence of multiple and isolated instances of user-space environments, leaving the hardware abstraction layer as well as the enforcement for process sandboxing to the shared kernel co-hosting them. Containers are lightweight software packages that contain all the dependencies required to execute the contained software application. These dependencies include things like system libraries, external third-party code packages, and other operating system level applications. The dependencies included in a container exist in stack levels that are higher than the operating system. Containers can be packaged into images, which are made up of layers. A new image can be built by adding a new layer on top of a previously existing image, making development more efficient and enabling possible bandwidth savings when transferring images across a network. Multiple versions of the same image can be created by using tags, which work as labels that can be used to mark versions (v1.2, v1.3) or mark an image as a preliminary version. When discussing services deployed at the edge, containers are almost always the preferred technology thanks to their lightweight properties.

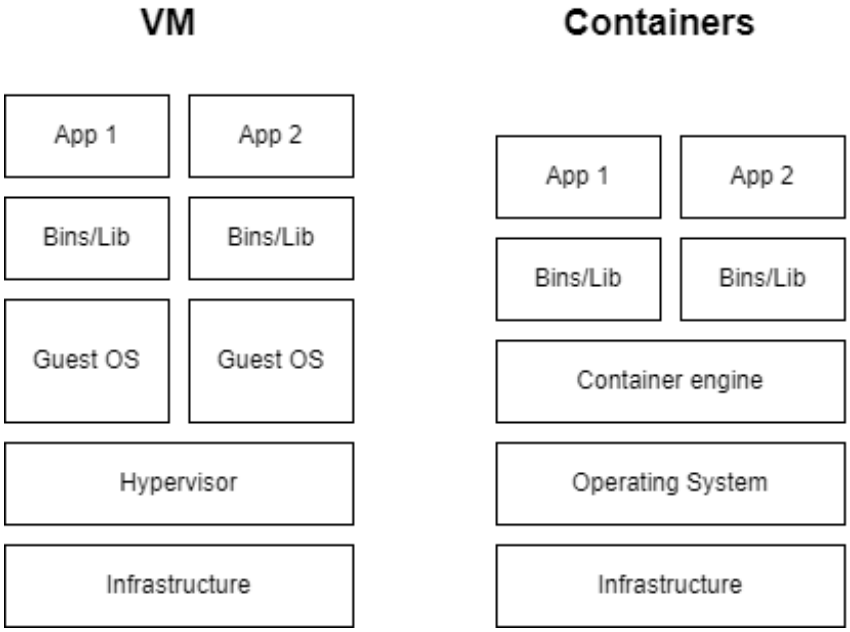


Figure 2.1: Difference between containers' and VMs' stack

2.1.1 The Open Container Initiative

The Open Container Initiative is an open governance structure founded by Docker and other leaders in the container industry in 2015 and is tasked with the creation of open industry standards around container formats and runtimes. More specifically, the OCI currently contains three specifications:

1. the Image Specification (image-spec) describes how to create an OCI image

2. the Runtime Specification (runtime-spec) describes how to run an OCI image
3. the Distribution Specification (distribution-spec) was introduced more recently to standardize the distrubution API of containers

2.2 runc

`runc` [3] is an open-source command-line tool that provides a standardized interface for spawning and running containers according to the Open Container Initiative (OCI) specification. It was originally created as a component of Docker, but it eventually developed into its own project and was donated to the OCI. `runc` is designed to be a low-level, lightweight tool that can be used to run containers on a variety of operating systems, container runtimes and orchestrators that support the specification, improving interoperability and portability. Kubernetes is one of the most popular among these orchestrators and uses `runc` to manage the lifecycle of containers. The features of `runc` include:

- support for a range of container configurations and settings, including resource constraints, network settings, and mount points;
- security, including seccomp, which limits the system calls that a container can make, and AppArmor or SELinux, which can be used to restrict container access to system resources;

`runc` can also be used directly by developers and administrators who need to run containers from the command line or integrate container management into their own scripts or tools.

2.3 CRIU

CRIU (Checkpoint/Restore In Userspace) [4] is a Linux software that can freeze a running container (or an individual application) and checkpoint its state to disk. The data saved can be used to restore the application and run it exactly as it was during the time of the freeze. CRIU is designed to be compatible with a variety of Linux kernels and architectures. It leverages Linux features such as namespaces and cgroups to provide a comprehensive and reliable process migration solution. It is currently integrated into all of the most relevant container runtimes available, such as LXC, Docker and Podman. Leveraging the kernel interface *ptrace*, CRIU seizes the process and injects parasite code to dump the memory pages of the process into image files from within the process's address space. To perform this step, CRIU first pauses the target process, injects the parasite code and releases it. The process now runs together with the injected code, which acts as a daemon waiting for commands from the main CRIU process. The main CRIU process can at any point instruct the parasite code to dump the states of the target process to disk and remove itself afterwards. The original process is unaware of all of this and CRIU gives the users the option of either killing it once the dump is complete or leave it running. The checkpoint image can be used to restore the process in the same or

a different machine. CRIU uses the *fork* system call to spawn the checkpointed process and its children processes and then restores all resources (e.g., open files, namespaces, and sockets) from the image that it previously read. The restoring process requires the PID of the original process (and its children processes) to be available on the host machine. Otherwise, the restoring fails at the very beginning of the procedure.

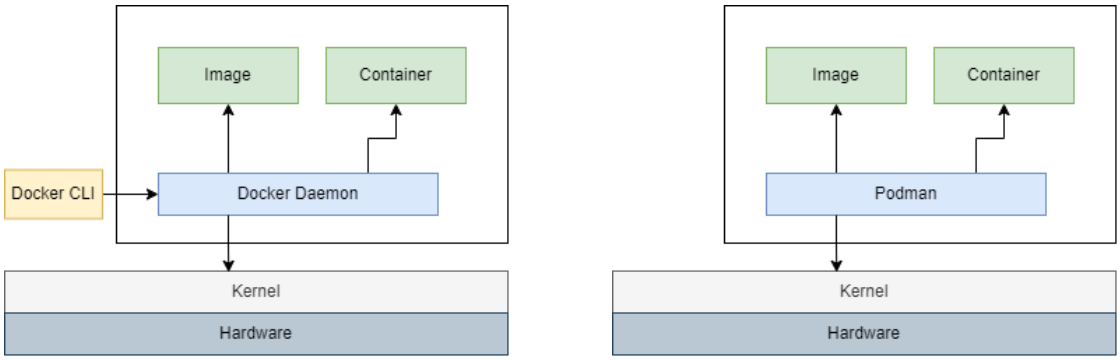
One of the CRIU features that is relevant to our case study is the ability to save and restore state of a TCP socket without breaking the connection thanks to the *tcp-established* option, which instructs CRIU to collect, along with the internal state of the container, the information related to the currently active TCP connection, thus allowing for a successful restoration of the TCP connection state during migration. Practical applications and limitations of this option will be discussed in a later chapter.

2.3.1 Containerization Technologies

A number of technologies are nowadays applied for the needs of the generation and application of containers, among them Docker, LXC, and Podman. Docker [5] stands out as the most extensively adopted container technology that is directly managed by the host kernel. Docker containers are configured through a Dockerfile, which comprises command-line interface (CLI) instructions and initial tasks. Docker images are generated using these Dockerfiles, encompassing all the executable source code, libraries, and dependencies required to instantiate Docker containers. Images can be created from scratch, downloaded from Docker Hub or created by building upon an already existing image. These images are immutable and consist of multiple layers, with new layers added at the top whenever modifications are introduced using specific Docker commands.

Like Docker, Podman [6] is an open-source, Linux-native tool by RedHat designed to develop, manage, and run containers and pods under the Open Container Initiative (OCI) standards. It is one of a set of command-line tools designed to handle different tasks of the containerization process, that can also function with any OCI-compatible container engine. Podman has a different conceptual approach to containers. Similarly to Kubernetes, Podman spawns container "pods" that work together. Pods are useful to organize separate containers under a common denomination to manage them as single units. One notable feature of Podman is its ability to run containers as either root or rootless. Due to its robust isolation capabilities and user privilege management, Podman offers enhanced security compared to other container technologies. When running a container, Podman further enhances security by implementing an additional isolation layer utilizing namespaces. Another defining feature of Podman is that it is daemon-less. A daemon is a program running in the background to handle services, processes, and requests with no user interface. Podman is a unique take on the container engine, as it doesn't actually depend on a daemon, but instead launches containers and pods as child processes. This difference is illustrated in figure 2.2. Podman also supports a CLI interface that is fully compatible with Docker, which enhances its ease of implementation in already existing projects. Among the many off the-shelf container engines, Podman is the one featuring the strongest integration with CRIU, by directly leveraging its APIs and, thus, effectively

supporting container migration at the microservice layer.



(a) Docker (b) Podman

Figure 2.2: Container engine architectures

Several studies have compared the performance of these technologies in common applications within the edge computing scenario. A very extensive work has been done by the authors of [7], who conducted a detailed performance benchmarking analysis of the containers measuring key performance indicators such as receiving time, waiting time, processing time, memory, and CPU usage and comparing them to the same applications running natively on the system. Their analysis involved Docker, Podman and Singularity containers. Singularity [8] is a container technology designed primarily for high performance computing. Instead of relying on synthetic benchmark tools, they developed a set of applications to benchmark image data and stream data performance and more accurately reflect IoT-Edge scenarios. They provide a highlight of the strengths and limitations of each container technology to aid the choice of the most suitable container technology for each specific edge computing use case. Their results for the wireless connection scenario are shown in figures 2.3, 2.4 and 2.5. The authors report better CPU and RAM utilization for Docker containers in most applications, with the lowest waiting time of approximately 0.9 seconds, comparable to native performance. In terms of processing time, Docker excels in Car detection (0.12 seconds), while Singularity and Podman outperform Docker in Object detection. A complementary work has been done in [9], in which the authors propose a mathematical model to measure and compare disk performance of the same application run natively, on Docker and on Podman containers. Their results show that while Podman has technically the best performance for all the tested workloads, the difference with Docker is so minimal to be negligible. They also highlight how the overload of using a container is very small with respect to native performance.

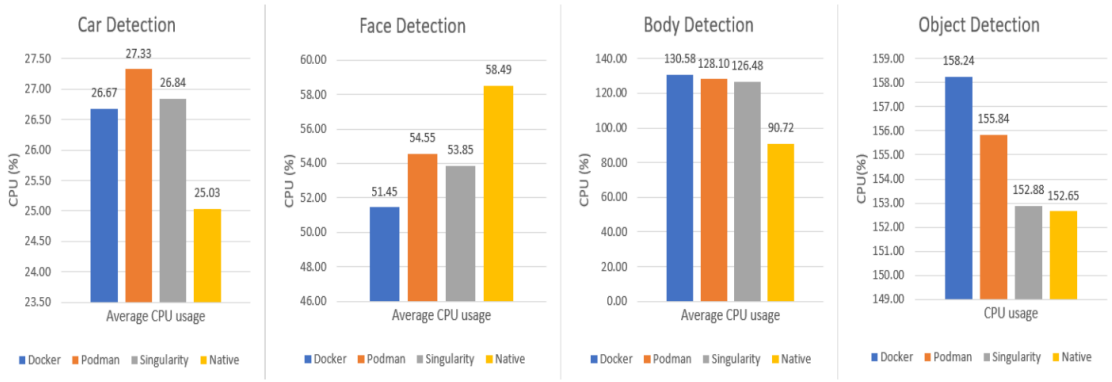


Figure 2.3: Average container CPU usage in wireless edge scenario [7]

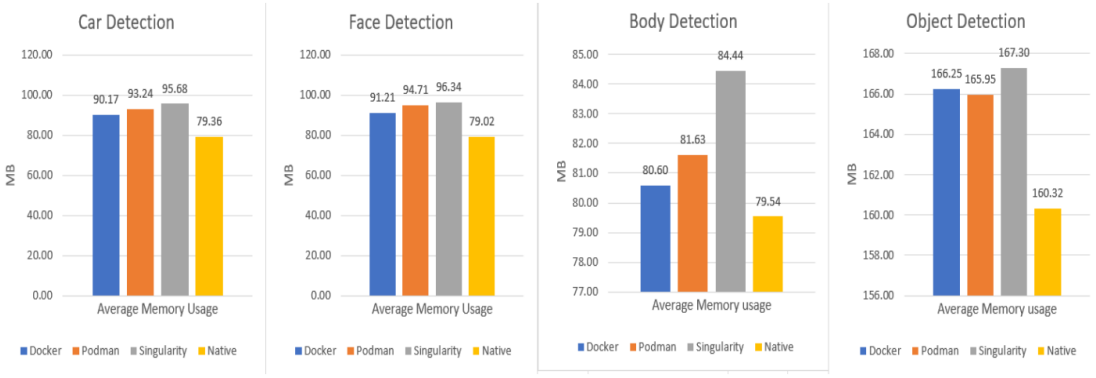


Figure 2.4: Container RAM usage in wireless edge scenario [7]

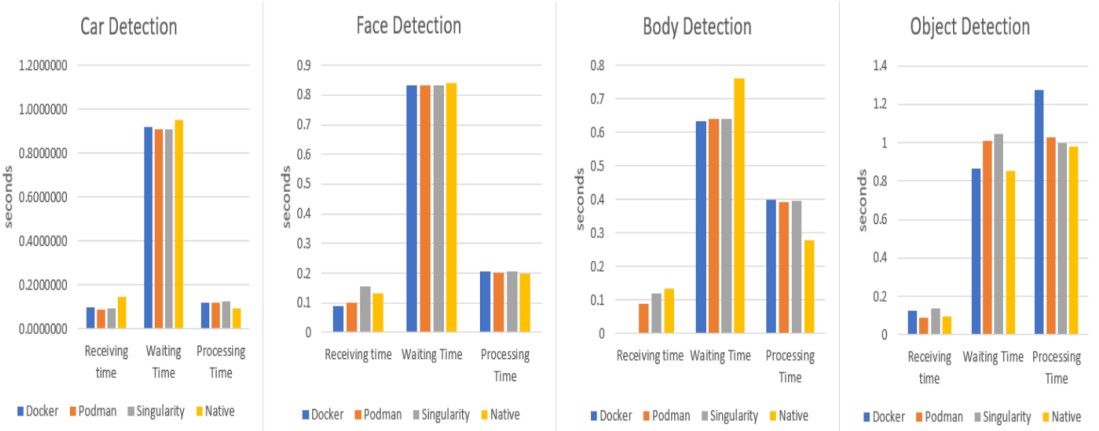


Figure 2.5: Average receiving, processing, and waiting time in wireless edge scenario [7]

Chapter 3

State of the Art Techniques for Stateful Migration

3.1 Edge Computing

Cloud computing and edge computing are two different computing paradigms that have become essential in recent years. While cloud computing is focused on centralizing computing resources and providing access to these resources over a network, edge computing aims to distribute computing resources closer to the edge of the network. The shift from cloud computing to edge computing has been driven by several factors, including the proliferation of mobile devices, the growth of the Internet of Things (IoT), and the increasing demand for real-time processing of data. In cloud computing, applications and data are typically hosted in a centralized data center, and users access these resources over the internet. Cloud computing provides a scalable, flexible, and cost-effective way to deliver computing resources, but it can also suffer from issues related to latency, network congestion, and data privacy. Edge computing, on the other hand, involves deploying computing resources closer to the edge of the network, typically at the edge of the enterprise network, the access network, or the device itself. This can help reduce latency, improve data security, and enhance the real-time processing of data. Edge computing is particularly useful in scenarios such as in the case of IoT devices or mobile devices, where data is generated at the edge of the network. By processing data closer to the source, edge computing can reduce the amount of data that needs to be transmitted over the network, improving efficiency and reducing latency. The shift from cloud computing to edge computing is not a binary choice, however, and both paradigms can coexist and complement each other. Hybrid cloud architectures, for example, can combine the scalability and flexibility of cloud computing with the low latency and real-time processing capabilities of edge computing.

Fog and edge computing are conceptually similar and the two terms in practice are often used interchangeably, but there are some nuanced significances that separate them from

each other. The main difference lies in where the intelligence is placed [10]. Fog computing envisions a layer at the local area network (LAN) level, where data passes through fog gateways where it is then transmitted to sources for processing. In MEC the computing power is located where data is generated. Edge servers and storage are installed on a device to collect and process data produced by sensors within the device. In our case the geographic scale at which we operate should not impact our approach to the solution, so we refer to edge as a general computing entity closer to the data producer that a cloud data center.

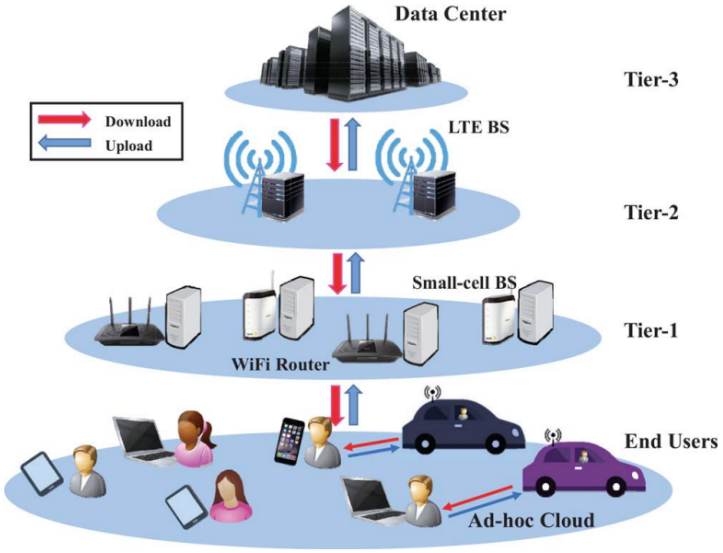


Figure 3.1: Reproduced from [11]

While edge computing has several benefits, there are also some challenges and limitations associated with this computing model [1]. Some of the problems with edge computing include:

- Limited resources: Edge devices have limited computing resources, such as processing power, memory, and storage. This can limit the type and complexity of applications that can be run on the edge devices.
- Security: Edge devices may be more vulnerable to security threats, as they are often located in remote and unsecured locations. Also, physically securing a large number of distributed devices can be challenging and complex.
- Data management: Managing data on the edge can be challenging, especially when dealing with large amounts of data. The data must be processed, analyzed, and stored in real-time, and ensuring data consistency and accuracy can be a challenge.
- Maintenance: Edge devices require regular maintenance, such as software updates and security patches. Maintenance can be difficult in distributed environments

where devices are located in different locations and environments.

To address these challenges, several initiatives have been undertaken to develop standardized architectures, interfaces, and protocols for edge computing. For example, the OpenFog Consortium and the Industrial Internet Consortium have developed reference architectures and guidelines for edge computing, while the EdgeX Foundry has developed an open-source framework for edge computing.

3.2 Stateless vs Stateful Migration

A stateless process or application can be understood in isolation. There is no stored knowledge of or reference to past transactions. Each transaction is made as if from scratch for the first time. Originally, containers were built to be stateless, as this suited their portable, flexible nature. But as containers have come into more widespread use, people began containerizing (redesigning and repackaging for the purposes of running from containers) existing stateful apps. This gave them the flexibility and speed of using containers, but with the storage and context of statefulness. For this reason stateful applications sometimes look a lot like stateless ones and vice versa. For example, an app can be stateless, requiring no long-term storage, but allowing the server to track requests originating from the same client by using cookies. Stateful container migration is the process of moving a containerized application from one host or environment to another, while preserving the application state and configuration. In edge computing, container migration can be used to optimize the use of computing resources and improve the performance of applications. There are several reasons why container migration is important in edge computing. First, edge devices may have limited computing power, and container migration can help balance the workload across multiple devices, making more efficient use of available resources. Second, container migration can improve the reliability and availability of applications in the event of a hardware failure or network outage, which is more likely since locations are harder to secure than cloud data centers. Finally, container migration can reduce the response time for applications. There are several challenges associated with container migration in edge computing, mostly linked to the variance in network conditions outside the controlled environment of a cloud infrastructure. For example moving containers from one edge device to another can consume significant network bandwidth, which may be limited or unreliable in edge environments. Also the time required to move a container from one edge device to another can impact application performance and increase latency, especially if the migrated container is being used to handle real-time data. Moreover moving containers between different edge devices can raise security concerns, especially if the destination device is not trusted or is located in a different network. Finally ensuring compatibility between different edge devices and container platforms can be challenging, especially if the devices are running different operating systems or container runtimes.

Stateless migration is a process that involves only two steps:

1. a new instance of the container is started at the destination

2. the old instance is closed

Stateful migration aims instead to make both volatile and persistent states available at the destination once migration is completed. We here give an overview of the state of the art techniques for stateful migration and discuss their advantages based on results presented in other works.

3.3 Cold migration

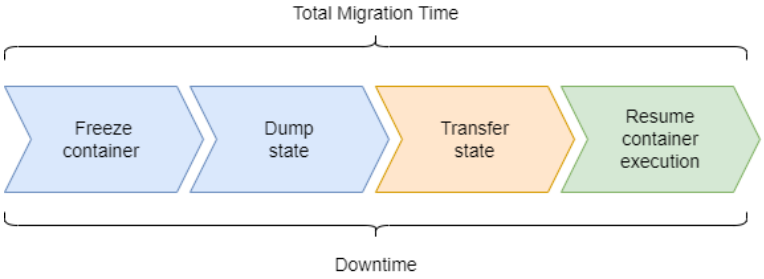


Figure 3.2: Cold migration steps

Cold migration consists in the following steps:

1. freeze the container to ensure its state doesn't change
2. dump the frozen state and transfer it
3. resume the container at destination once the state is available

Cold migration is characterized by a very long downtime that coincides with the total duration, however one advantage is that as the state is transferred only once the amount of data sent and the total migration time are both lower than other solutions.

3.4 Live Migration

Alternatively to cold migration, live migration aims to transfer the state of the container without the need to freeze it for the whole duration. By freezing the container only for a small fraction of the total migration time the downtime is minimized, and when this downtime is not noticeable by the end user live migration is said to be "seamless". Live migration can happen through one of three techniques: pre-copy, post-copy and hybrid.

3.4.1 Pre-Copy Migration

Pre-copy migration is so called because it transfers most of the state before freezing the container for a final dump and state transfer, after which the container runs on the destination node. It is also known as iterative migration, since it may perform the pre-copy phase through multiple iterations such that each iteration only dumps and

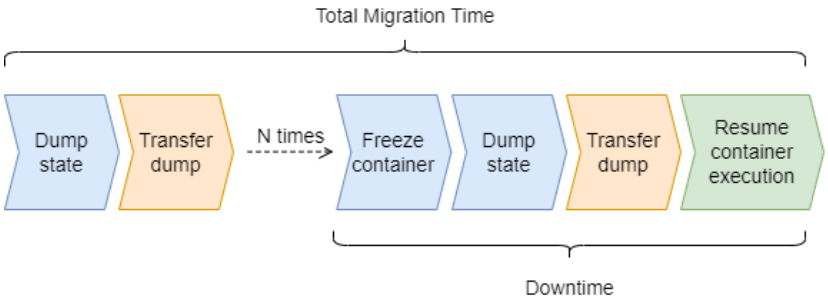


Figure 3.3: Pre-copy migration steps

retransmits those memory pages that were modified during the previous iteration (called dirty pages). The modified memory pages are called dirty pages. The first iteration dumps and transfers the whole container state as it is done in cold migration but without stopping the execution. As the number of iterations increases, the amount of time it takes to transfer new dirty pages converges to a shorter duration. Typically after a set number of iterations the container is frozen and one last dump is transferred to the destination, where the new instance is started.

The main difference between cold and pre-copy migrations lies in the nature of their dumps. The one and only dump in cold migration consists in the whole container state and thus always includes all the memory pages and the execution state. In pre-copy migration there is one initial pre-dump phase during which the whole state in transmitted, but the following iterations only include those memory pages that were modified during the transmission of the previous dump, together with the changes in the execution state. As such, downtime for pre-copy migration should be in general shorter than that for cold migration because less data are transferred while the container is stopped. This also means however that the amount of data that is transferred depends on how fast the state is changing, resulting therefore in a non deterministic downtime. More formally, we define dirty page rate the speed at which the hosted service modifies the memory page. The amount of data transferred must also be weighed against the throughput, since a longer transfer time for the pre-copy phase results in a larger number of dirty pages. As a final remark, we highlight that, unlike cold migration, pre-copy migration might transfer each memory page several times, with possible negative consequences on the overall amount of data transferred during migration and thus on the total migration time.

3.4.2 Post-Copy Migration

Post-copy migration contrary to pre-copy, first suspends the container on the source node and copies the execution state to the destination so that the container can resume its execution there. Only after this step (post), it copies all the remaining state, namely all the memory pages. Depending on how this second step is performed, post-copy algorithms can be categorized into three variants. The only variant currently implementable using the tools provided by CRIU is the one called post-copy migration with demand paging

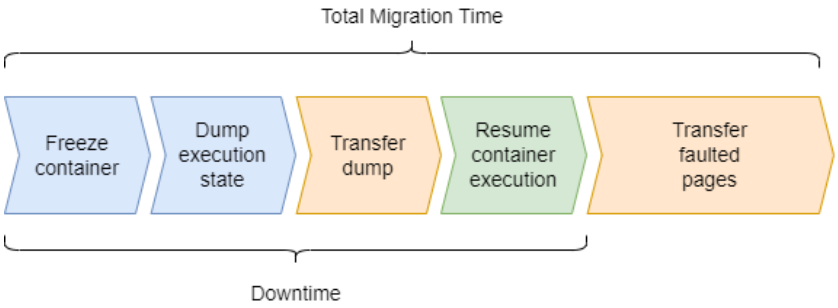


Figure 3.4: Post-copy migration steps

variant, better known as lazy migration. With lazy migration, the resumed container tries to access memory pages at destination, but, since it does not find them, it generates page faults. The outcome is that the lazy pages daemon at destination contacts the page server on the source node. This server then "lazily" (i.e., only upon request) forwards the faulted pages to the destination.

Unlike pre-copy, post-copy transfers each page only once resulting in an amount of transferred data comparable to that of cold migration. Moreover since dump in post-copy migration is simply the execution state and does not contain any dirty memory pages, downtime is irrespective of the page dirtying rate featured by the container-hosted service and of the overall amount of data that needs to be transferred. Post-copy migration however is affected by two major problems: (i)page faults degrade service performances, as memory pages are not immediately available at destination once the container resumes, to a point where the technique may become unviable by latency-sensitive services; (ii)during migration, the overall up-to-date state of the container is distributed between both the source and the destination node (before completion of post-copy migration, the source node retains all the memory pages, but some of them may be out-of-date because they have already been copied at destination and modified by the resumed container), whereas approaches like cold or pre-copy migrations retain the whole up-to-date state on the source node until the termination of the migration process. Therefore in case of destination node failure during migration, it may be no longer possible to recover the up-to-date state of a post-copied container.

3.4.3 Hybrid Migration

Hybrid migration aims to combine the two previously illustrated techniques for live migration in order to mitigate their shortcomings. The first two steps of hybrid migration are the same as those of pre-copy migration: a pre-dump of the whole state and its transmission at destination while the container is still running on the source node. After that the container is stopped and a dump is generated that includes the modifications in the execution state that occurred during the pre-copy phase. Once the dump is transferred to the destination node, the container can be restored. At this step, the destination node

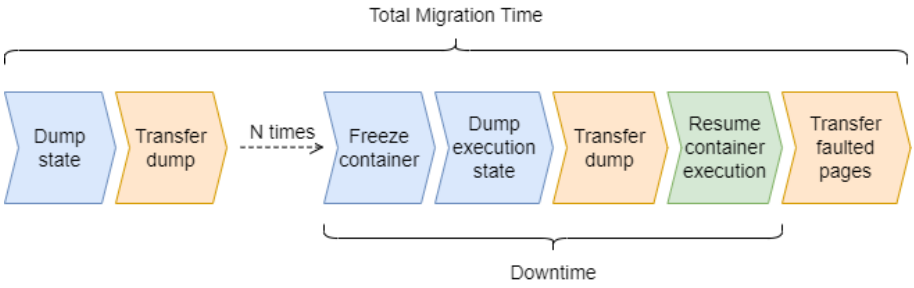


Figure 3.5: Hybrid migration steps

has the up-to-date container execution state along with all the memory pages. Nonetheless, some of them were dirtied during the pre-copy phase. As a result, the last step in hybrid migration consists in the page server at source lazily transmitting the dirty pages to the lazy pages daemon on the destination node. It is worth noting that the number of memory pages that are lazily transmitted in hybrid migration is generally less than that of post-copy migration since only the dirty pages are transferred. From now on, we will refer to these dirty pages as faulted pages, in line with the name used in post-copy migration to indicate the data transferred in this last phase.

Like for pre-copy migration, hybrid migration is also affected by the page dirtying rate and the amount of data that are transmitted during the pre-copy phase. However, these factors should influence the total migration time for hybrid migration but not the downtime, as they alter the number of faulted pages. Hybrid migration is affected by the same two drawbacks of post-copy migration, namely service quality degradation and possibility of data loss in case of failure, though to a lesser degree.

Chapter 4

Multipath Protocols applied to Seamless Stateful Migration

4.1 General Notions on Multipath Protocols

Multipath routing protocols create multiple routes from the source to the destination instead of the conventional single route. They are widely employed and researched thanks to their ability to achieve load balancing, which helps during times of network congestion that may arise due to bursty traffic within the network, and being more robust to link failure, since in case of link disconnection within an active route they require less effort for the alternate route discovery. They can also be used to improve total connection bandwidth by multiplexing the data stream over several network interface cards (NIC). While implementation details differ between protocols, they all share some features:

- multipath path management, the ability to initiate and manage several connections within the same multipath connection (i.e. subflows)
- multipath scheduling, the ability to distribute packets over different paths following a certain policy
- multipath congestion control, the ability to detect network congestion and adjust the sender rate accordingly (as in the single path case)
- reliable transfer, for loss detection and loss recovery

In particular, the multipath congestion control should have the following goals:

- Fairness to single-path protocols: if several sub-flows are running alongside a regular connection over a bottleneck link, the multipath protocol should not be able to get more throughput than the regular connection
- Incentive to deploy: The performance of the sub-flows of a multipath session should

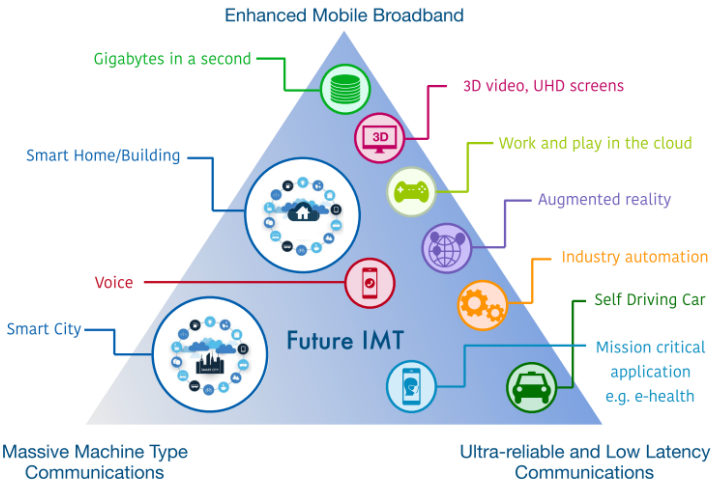


Figure 4.1: 5G services and corresponding reference use cases. Reproduced from [11]

at least match the throughput of a regular connection

- Load balancing: the paths used to send data by the active sub-flows should be the ones which are experiencing less congestion than the others

4.1.1 Role of multipath protocols in 5G architecture

The 5th generation of mobile communications (5G) has greater aims than its predecessors, whose efforts were mostly targeted towards achieving a wireless connection closer to a wired one. The International Telecommunication Union (ITU) has defined three major performance aspects that are central in 5G [11]: enhanced Mobile Broadband (eMBB), Ultra-Reliable Low-Latency Communications (URLLC), and massive Machine Type Communications (mMTC). The goal of eMBB is to meet the people's demand for an increasingly digitally connected lifestyle, enabling services that have high bandwidth requirements such as high definition video streaming, and virtual/augmented reality (VR/AR) applications. URLLC focuses on latency-sensitive and high-reliability services such as assisted and automated driving, remote robotics, and mission-critical applications. mMTC is necessary to accommodate the increasing presence of IoT in applications such as smart cities. To fulfill these requirements 5G envisions the usage of new technologies such as millimeter waves, Massive MIMO, Network Slicing, Software defined Networking (SDN), Network Function Virtualization (NFV), and Multi-access Edge Computing (MEC). We focus on this last one in particular, enabled by the fact that today's commercial devices often come equipped with several Network Interface Cards (NIC) which can be and are being exploited to improve resiliency to failure and total bandwidth.

Access Traffic Steering, Switching and Splitting (ATSSS) is a technology that leverages

multipath transport protocols to deliver the following functionalities: (i)steering: enables the selection and use of an access network for a data flow; (ii)switching: allows to maintain service continuity while redirecting all traffic of an ongoing data flow from one access network to another; (iii)splitting: enables the splitting of the traffic of a data flow across multiple access networks, so that some traffic of the data flow is transferred via one access and some other traffic of the same data flow is transferred via another access. In the Technical Specification 23.501 (Release 16) [12], the 3GPP (3rd Generation Partnership Project) specifies an ATSSS architecture to support the switching between 3GPP access (e.g. LTE and 5G New Radio (NR)) and non-3GPP access networks (e.g. WiFi). ATSSS is useful both to eMBB, delivering increased throughput through concurrent transmissions, and to URLLC requirements, delivering low latency and high reliability through path redundancy. Given these conditions it's easy to foresee an increase in relevance of multipath protocols for a multitude of applications, which makes designing solutions around them more than reasonable.

4.2 Seamless migration employing standard TCP

Solutions employing standard TCP need to deal with the fact that this protocol maps connections to sockets identified by the IP address and port number of the local device (local endpoint), as well as the IP address and port number of the remote device (remote endpoint). A TCP connection can be migrated by leveraging the *tcp-established* feature of CRIU and the *TCP_REPAIR* option for the TCP socket. Thanks to the *tcp-established* feature, CRIU can collect, along with the internal state of the container, the information related to the currently active TCP connection, thus allowing for a successful restoration of the TCP connection state during migration. Meanwhile, when the *TCP_REPAIR* option is used, the TCP socket is switched into a special mode where any native TCP action performed on the socket has no effect allowing CRIU to checkpoint its state. This combination however does not work if the IP address changes after migration or if the microservice container is not able to directly reach the client when moved to the destination host (which happens quite frequently in the case of migration between distinct private networks).

The authors of [13] propose a network architecture named COAT that allows to preserve the existing TCP connection between the client and the migrated microservice, keeping track of the TCP connection states (thus avoiding reconnection procedures) and preserving all the data queued inside the TCP socket (thus preventing data losses) while being transparent to both sides. They do this through the usage of Virtual Extensible LANs (VXLAN) to establish an overlay network, over which users can easily define or change the behavior of virtual switches through the OpenFlow protocol. As shown in figure 4.2, two custom namespaces are created for the mobile user and microservice host respectively. By imposing an exact recreation of the microservice namespace at the destination host, the *TCP_REPAIR* option can be used to preserve the TCP connection since the same IP address can be easily replicated at the destination host.

The authors experimentally validate their proposed COAT architecture by evaluating its

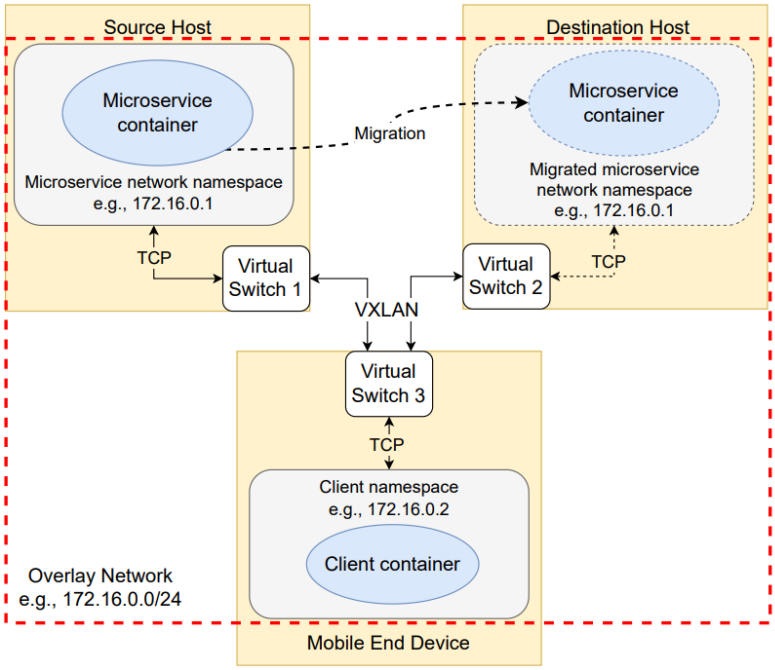


Figure 4.2: COAT network architecture [13]

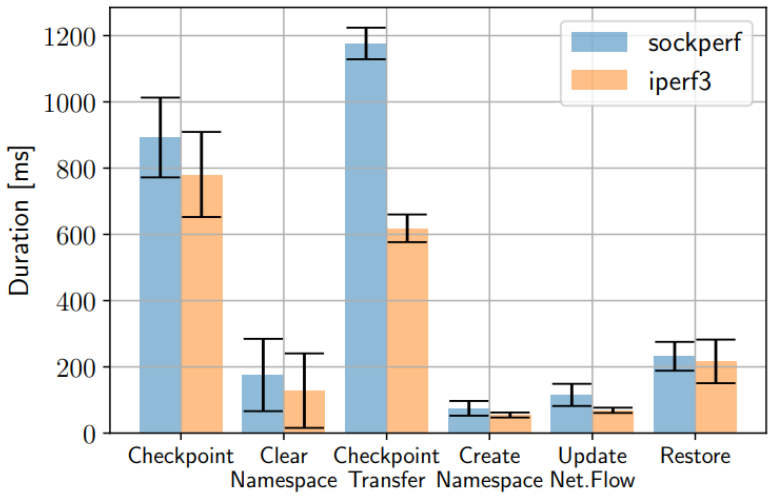
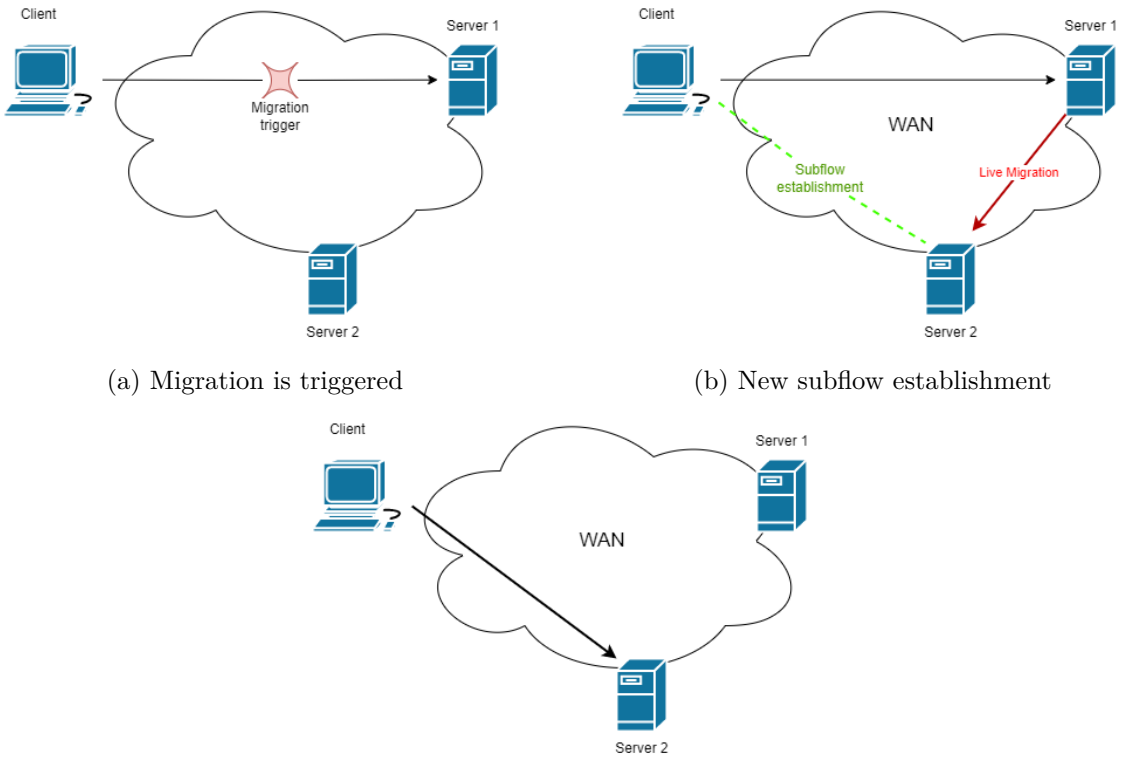


Figure 4.3: Duration of each step of the COAT migration process [13]

performance, performing migration of two microservices (*sockperf* and *iperf*) over virtual machines in the cloud. Figure 4.3 shows the actual duration of each migration step. The higher values for *sockperf* are due to the larger state size. They conclude that the COAT migration does not introduce any time overhead in the three most impactful migration steps (checkpoint, transfer, and restore) with respect to the original process and that the additional steps introduced by this solution amount to an increase on the total migration

duration of around 14%.

4.3 Model description



(a) Migration is triggered

(b) New subflow establishment

(c) End of migration

Figure 4.4: Steps envisioned in the proposed migration solution

In the described MEC context, we would like for the migrated service to be operative immediately after migration is complete.

These problems could be addressed by the features of multipath protocols described in 4.1. The particular solution we want to investigate is the one shown in figure 4.4 and consisting in the following steps:

1. migration is triggered
2. a new subflow is created between the client and the new container instance; note that this step will most likely require knowledge of the new IP address beforehand
3. once both subflow creation and migration are completed, the old container instance is shut down without warning. Thanks to the multipath path manager, the traffic is automatically steered towards the newly created subflow without the added latency due to the negotiation of a new connection

There are a number of considerations to be made on each of the described steps. These are meant as a reference when analyzing existing protocols, to see which problems they solve and what still needs to be worked out.

STEP 1

We have previously mentioned that the migration process must be triggered by some entity. The possible candidates to fill this role are:

- the user application client
- the edge application server
- the edge orchestrator

The choice may be case specific and we therefore do not investigate further on this topic, however we must design our solution taking into consideration the fact that any of these strategies should be compatible. Assigning this role to the orchestrator may simplify the process of acquiring the IP address of the destination edge container.

STEP 2

Within the subflow creation step there are several fundamental decisions that must be taken. Note that subflow creation can happen at any time during migration (namely before, while and after the container is transferred), but the choice must be made with awareness on the constrains imposed by the subflow establishment procedure of the chosen multipath protocol. These constraints should also be taken into consideration when choosing which entity is responsible for initiating this procedure. In this case, the possible solutions are:

1. The client, who is aware of the ongoing migration, receives information on the IP address of the destination server container and sends the subflow request. This is the simplest solution to implement and, as we will see in later chapters, is compatible with most protocols. It however requires an orchestrator/network manager that is aware of the migration and the destination IP address.

2. The source edge creates the subflow in place of the destination edge acting as a proxy and forwards the data until it is ready to be terminated. This solution is trickier, since issues are likely to emerge for path validation and endpoint authentication.

3. once the container is migrated to the destination server, the subflow is established directly from there before the old instance is terminated. This solution requires the client to be able to recognize subflow request packet from unknown address. To reap any kind of benefit in container downtime, the new container must be up and running before migration is complete.

4.4 MPTCP

MPTCP (Multipath TCP) is an extension to the traditional TCP (Transmission Control Protocol) that allows multiple paths to be used between two endpoints to provide increased bandwidth, improved resilience, and better performance. The MPTCP v1 protocol is defined in RFC 8684 [14]. While traditional TCP uses a single path to transfer data between two endpoints, which can result in congestion and limited bandwidth if the network path becomes congested or fails, MPTCP allows multiple paths to be used simultaneously, which enables the traffic to be distributed across different network paths and avoids bottlenecks. MPTCP uses a subflow mechanism to establish multiple paths between two endpoints, and each subflow is treated as a separate TCP connection (Figure 4.5). MPTCP also includes congestion control mechanisms that ensure fair sharing of the available bandwidth across all the subflows. The Linux MPTCP community develops and maintains the MPTCP v1 stack in the Linux kernel (v5.6 or later) and associated userspace tools and libraries. A Linux implementation is composed of 4 main blocks, described in [15]: (i)the meta socket, the central abstraction of each MPTCP connection, (ii)the path-manager, which decides on the creation and removal of subflows, (iii)the congestion control, responsible for the congestion window sizing based on multiple algorithms, and (iv)the scheduler, whose main task is to transparently split data among the subflows. Implementing MPTCP does not consist in simply replicating the TCP connection for each subflow. New options have to be added to the protocol, considering that each packet can take different paths with the related changes on the standard TCP mechanisms (i.e. out-of-order data arrival at the receiver, middleboxes able to modify packets).

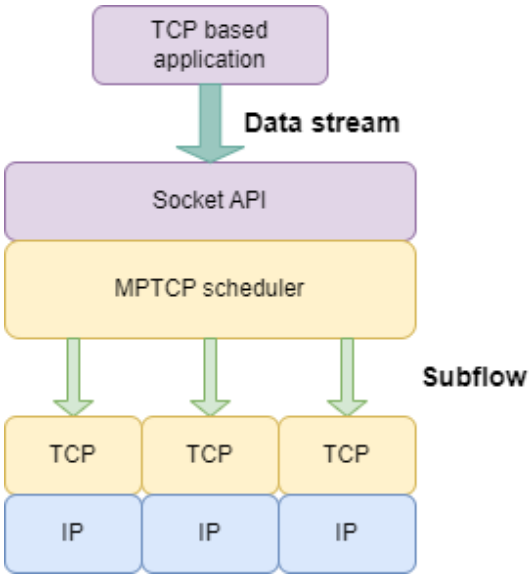


Figure 4.5: MPTCP stack

The first connection establishment (figure 4.6) is performed through the three-way handshake (ACK - SYN ACK - ACK packets) just like for standard TCP. When the client

has an additional interface, it will notify the server of its additional IP address with an ADD_ADDRESS option over the established subflow, then send another SYN packet with a JOIN option to the server's IP address. This method is used to avoid problems with Network Address Translators (NATs) filtering out packets that come from unknown addresses. End-hosts are recognized not by means of their IP address but through the connection associated to the established meta socket. Each MPTCP subflow behaves as a TCP flow so, after the 3-way handshake, each subflow maintains its congestion window and retransmission scheme during data transfer. As largely discussed in the literature [16] [17] [18], the choice of congestion avoidance algorithm and scheduler profoundly impacts the overall performance. As of the writing of this paper, the Minimum RTT algorithm is the current default scheduler in the MPTCP Linux Kernel. It prioritizes assigning packets to the subflow with the lowest Round Trip Time (RTT) and non-exhausted congestion window. It proves to be beneficial in heterogeneous networks to exploit the best link available, which can also be a negative thing since it may cause congestion. The authors of [19] propose and validate an analytical model to predict performances of MPTCP in the case of two subflows sharing a bottleneck in both slow start and collision avoidance phases of the individual TCP flows.

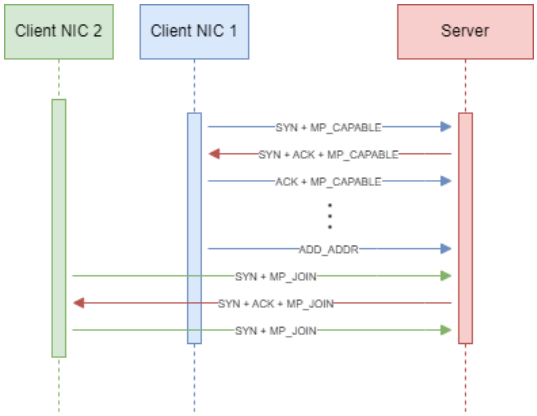


Figure 4.6: MPTCP handshake and subflow establishment

MPTCP is used in a variety of applications, such as video streaming, file transfer, and live media, where high bandwidth and low latency are critical. However, not all network environments and applications may be suitable for MPTCP, and it may require some network configuration and management to ensure proper operation. In [17] Hurtig et al. prove that network interfaces with asymmetric capacity and delay typically result in a poor user experience when coupled with low latency communication attempts. They propose two novel scheduling techniques as possible solutions: the BLocking ESTimation (BLEST) scheduler and the Shortest Transmission Time First (STTF) scheduler. The authors in [16] also investigate the problems in adopting a multipath approach, concluding that the current MPTCP is not Pareto optimal, meaning it can reduce the throughput of other TCP connections without bringing any benefit while being excessively aggressive toward TCP users. We also need to note that, similarly to TCP, MPTCP is not secure by default. It has therefore no option but to rely on additional security protocols on top

of it, which however leave some vulnerabilities during the establishment of a new subflow [20].

4.4.1 MPTCP in Linux Kernel

In this section we describe the MPTCP implementations that in our experience are more easily available and give an overview of how to use them. A very popular MPTCP linux kernel is available in the dedicated website [21]. This implementation is often used in papers because it forces the creation of MPTCP sockets even by applications which do not support it natively, allowing accurate testing and performance measurements without the need of full integration with the examined application. It is a custom kernel based on Linux 5.4 that allows to configure the MPTCP parameters through a set of available options, modifiable by means of *sysctl*. The available MPTCP options are:

- `net.mptcp.enabled`: This option enables the use of MPTCP when it is set with the value 1, otherwise it can be disabled by setting the value to 0.
- `net.mptcp.mptcp_syn_retries`: This option configures the number of retransmitted SYN with the MP_CAPABLE option, after which the MP_CAPABLE option will stop being advertised. The default value is 3. This option exists to handle middleboxes that drop SYNs with unknown TCP options.
- `net.mptcp.mptcp_checksum`: This option enables/disables the use of MPTCP checksum when set with the value 1 or 0.
- `net.mptcp.mptcp_path_manager`: Allows to choose between the two compiled path-managers. This structure is necessary for the creation of new sub-flows and also to advertise alternative IP addresses through the ADD_ADDR option. The choices available are: (i) default, where no subflows are established at the start of a connections, (ii) fullmesh, a subflow is established between each possible pair of advertised interfaces.
- `net.mptcp.mptcp_scheduler`: Allows to select the preferred scheduler. The ones available off the shelf are minimum RTT, round robin and redundant.

This implementation is unsuited for practical deployments for several reasons. The first one is that it requires the installation of a custom kernel, which limits the applications for containerized applications that are deployed on an already existing VM. Moreover, this kernel is not upstreamable because its TCP stack is subject to too many modifications. These modifications are targeted towards maximum MPTCP efficiency but come at the price of standalone TCP performance and ease of maintenance.

As already mentioned in a previous chapter, the linux kernel version of the MPTCP stack is maintained by the Linux MPTCP Upstream Project [22]. It fits upstream standards and is present within each new kernel release (currently 5.19). It is therefore the most likely version to be available on the edge VM. An application that wants to use MPTCP should do so by creating sockets using `IPPROTO_MPTCP` in the proto field (e.g.

socket(AF_INET, SOCK_STREAM, IPPROTO_MPTCP);). Legacy applications that are not integrated with MPTCP can be forced to create MPTCP sockets through the *mptcpize* command. This command is bundled within the MPTCP daemon, which is another project maintained by the community [23] that performs multipath TCP path management related operations in the user space. Routing and path management can be configured through the *ip* packet, which has a dedicated *mptcp* option. Some useful commands are:

- `sudo ip mptcp limits set subflow N`: this command sets the maximum number of subflow that a connection can establish to N
- `ip mptcp limits set add_addr_accepted N`: this command sets the maximum number of additional addresses that can be advertised within a connection to N
- `sudo ip mptcp endpoint add IP_ADDR dev ETH subflow`: this command creates a new endpoint that can be advertised by MPTCP connections. The final option specifies for what use the address is advertised and can assume the values subflow or backup.

4.4.2 MPTCP applied to migration

MPTCP's application to MEC scenarios has been subject of many studies. In [24] the authors give an overview of how live migration can be managed in a cloud environment, using the multipath features mostly for the increased bandwidth. They discuss in detail how a faster data transmission drastically improves migration time for pre-copy migration: less time to transfer a dump means a higher dump frequency, which results in smaller deltas and faster convergence. The authors of [25] show how the decisions made by the MPTCP scheduler based on sender perceived RTT are not suited for edge environments, and suggest a Receiver Assisted MPTCP (MPTCP) that incorporates both sender and receiver side last hop MAC characteristics. Very few studies however tackle the possibility of using MPTCP to address connection reestablishment downtime issues during stateful migration. The authors of [26] propose a solution for workload balancing that makes use of MPTCP to facilitate VM live migration between edge networks, but while they recognize the issue of IP address changes their algorithm simply assumes that the new IP's are known and that an additional subflow can be created preemptively between them. We find an interesting discussion about connection migration between hosts in [27], in which the authors discuss and implement an endpoint migration using what they call connection acrobatics. Their goal is to include middleboxes in the internet architecture in a deployable way, leveraging MPTCP and proposing a "sticky bit" upgrade to reduce connection redirection overhead. As the authors themselves acknowledge though, this upgrade is prone to exploitation by attackers who can use it to hijack connections.

They state that multipath TCP's address management mechanisms are sufficient to implement the needed redirection mechanisms, discussing also the case depicted in figure 4.7 which describes a case of mobility when middleboxes are involved. In their own words: "For redirection, the ADD_ADDR functionality can be used to redirect traffic as follows.

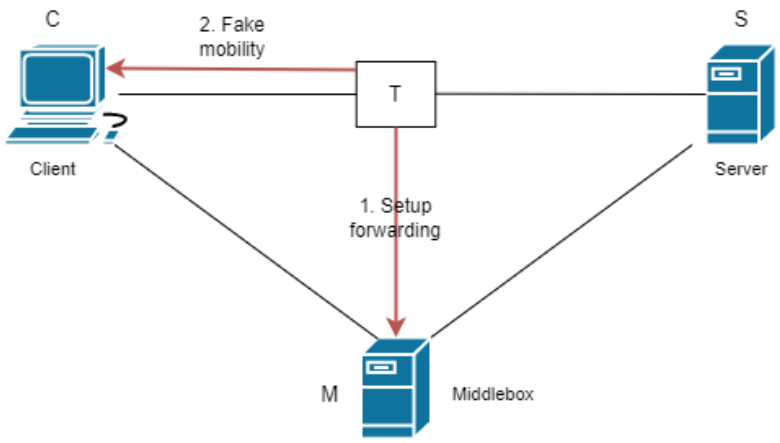


Figure 4.7: Case study of MPTCP migration involving a middlebox

Say an MPTCP connection is setup between C and S. T sets up a forwarding rule at M instructing it to proxy all traffic it receives from C. Then, T sends an ADD_ADDR message to C advertising M's address. As a result, C will send a SYN+MP_JOIN message to initiate the three way handshake. M receives the SYN+MP_JOIN message, it changes the source address to M and the destination address to S and forwards the message to S. An analogous processing is applied to subsequent packets of the three way handshake. The result is that Both C and S have a new subflow with M, which acts as an explicit middlebox for the MPTCP connection. T can now close the C-S subflow, effectively forcing the endpoints to use only the path via M. Once M sees the traffic, it can direct the traffic to S or D via another proxy by simply using ADD_ADDR and setting up the appropriate proxy rules". Their results show a total redirection time of around 200ms which is still comparable to the time it would take in our case to reestablish the connection from scratch, but since it also includes the time needed to transfer the MPTCP session to the new server in our case it would be partially absorbed by the migration process. The authors do not provide any details on their implementation, other than the fact that they had to modify the MPTCP stack of the MPTCP kernel implementation.

The authors of [28] report their solution for live service migration based on the MPTCP design described by combining [27] and the tunneling of [29], describing also the challenges they encountered during the implementation. Their solution performs the following steps, described in figure 4.8:

1. A tunnel is created between the source and destination hosts before the migration procedure is triggered, also routing for both traffic incoming from and traffic outgoing to the destination host server is prepared. Routing configuration at the destination host server minimizes the service down time, and is possible since no traffic is yet arriving at the destination.

2. The migration procedure is triggered.

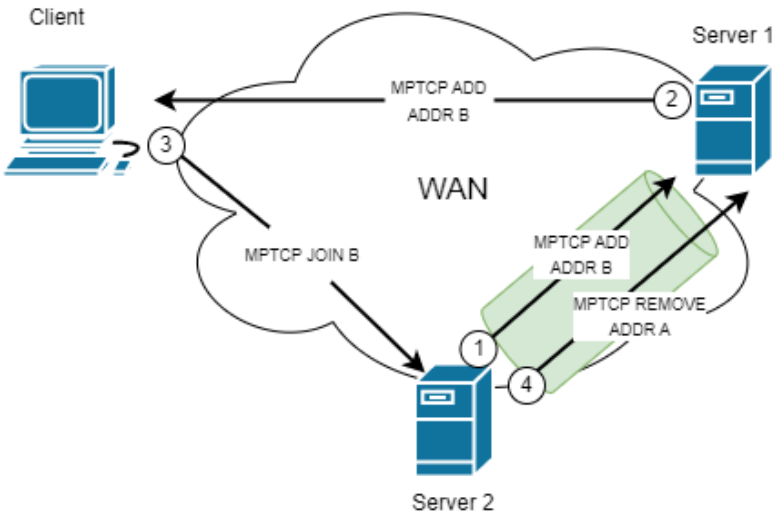


Figure 4.8: Case study of migration through the WAN by means of tunneling and MPTCP

3. As soon as the migration procedure is completed, routing entries are created at the source host server to forward traffic across the tunnel opened in the first step. Until this step is completed there is no connectivity towards the destination server, so speed is critical.

4. after the destination host server has gained connectivity, it creates new subflows towards the client that do not traverse the tunnel but go directly into the internet.

5. Once the new subflows are established, the old ones are shut down and resources at the original host server are released.

Their implementation is made to be used on VMs and is based on the reasonable assumption that each host server has two physical NICs: eth0 connected to a highly provisioned private network exclusive to data centers and eth1 connected to the public internet. The tunnel between the two servers and the migration leverage their eth0 interfaces, while direct communication with the client happens through eth1. This separation ensures the lowest performance degradation during the migration. Their solution does not implement any mechanisms to prevent packet loss during application downtimes, as they themselves report that uploads can experience loss rates of up to 35% in their environment during migration, compared to that of 1% for downloads, showing that this is because of packets arriving while the VM is suspended during migration and not due to the network itself. This solution looks very promising, as it leverages a very popular protocol in an elegant way to achieve (almost) seamless migration, however some considerations have to be made. The assumptions it is based on could pose a constraint when applying it to containers, as they not always have free access to the NICs of their host. While the option is available both for Docker and Podman containers (by using the `-network=host` option), some security concerns may arise especially if not all containers on that host

are from the same owner. The most significant issue with this solution is however the implementation: as the authors describe in detail, several non-trivial modifications to the network stack are required (such as to the ARP table and MTU size). This makes the solution incompatible with a standard off-the-shelf MPTCP kernel and is very likely to hurt its deployability.

4.5 MPDCCP

Datagram Congestion Control Protocol (DCCP) [30] is an unreliable transport layer protocol with a congestion control mechanism. It was designed to better suit the needs of multimedia streaming, which prefers timely data rather than complete data but is prone to congestion due to the high bandwidth required. Although DCCP is an unreliable protocol that does not retransmit lost packets, it still involves acknowledgement for the received packets. DCCP provides general support for various congestion control algorithms that can be viewed as pluggable modules, enabling link quality estimation. The DCCP standard does not specify one single congestion control algorithm, rather several congestion control profiles have been defined for DCCP (such as ID2 and ID3 specified in RFCs 4341 [31] and 4342 [32] respectively). There is a relatively limited body of academic research on the DCCP protocol, mostly investigating the differences between DCCP and TCP, or performance of streaming video applications over DCPP. Just like MPTCP, MPDCCP is a set of additional features on top of DCCP which operates at the transport layer and aims to be transparent to both higher and lower layers. The protocol stack, shown in figure 4.9, is also very similar to MPTCP. MPDCCP is designed to be used by applications in the same way as DCCP with no changes to the application itself.

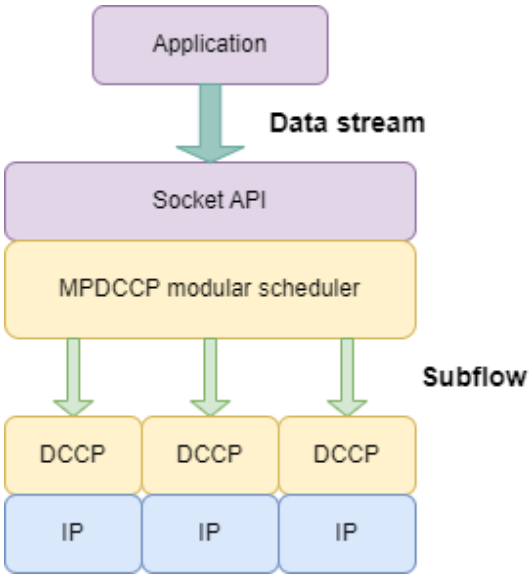


Figure 4.9: DCCP stack

The MPDCCP architecture consists in two modules: a packet scheduler and a packet

reordering module. The packet scheduling module distributes the data over the available DCCP flows. It takes decisions based on the available information on such flows, which include the current estimate of the DCCP channel state. After the selection of a DCCP flow, the current packet to be scheduled for transmission is encapsulated into the flow. Simple variants of the scheduling algorithm design can be chosen to fulfill different needs. Algorithms include the round robin scheduler, which equally shares packets between the available paths, the fixed ratio scheduler, which uses fixed weights to specify the ratio of packets scheduled on certain paths, or the cheapest pipe first packet scheduler, where the operator can assign a cost value to each network path and priority is given to the cheapest one. The packet reordering module has an advantage over the packet scheduling module as it does not rely on average channel feedback to make scheduling decisions to prevent out-of-order packet delivery. Packet reordering is instead based on monitoring the received packets, buffering out-of-order packets and delaying their delivery to the application for a pre-defined timing threshold. This is based on the expectation that during the buffering time the missing packets will arrive and fill the gaps. After the threshold has expired the buffered packets are always delivered to the application, ordered or not, and it is up to the application to manage the out of order reception. This concept limits the buffering delay and presents an advantage in comparison to the MPTCP-based framework, where performance is greatly worsened in case of large latency variance across paths since retransmissions may be requested for the delayed packets, resulting in increased overall delays for unreliable traffic. Two new options are introduced in the MPDCCP header in order to set a more appropriate threshold: one to convey the RTT information and one to include packet sequencing information. This sequencing is in addition to the standard DCCP sequencing to facilitate reordering at the receiver.

4.5.1 MPDCCP applied to migration

Multipath DCCP is not widely used in practice, and there are very few applications that are known to use it. One reason for this is that it requires support from both the client and server, and currently, there are very few implementations of Multipath DCCP available. As some researchers and developers continue to investigate the potential use cases for Multipath DCCP and explore its performance and suitability for different types of applications, it is possible that as the need for more efficient and reliable network communication grows, Multipath DCCP may become more widely adopted in the future. However MPDCCP is mostly overshadowed by MPTCP, which is widely deployed and researched, so it is not a good candidate for our purposes.

4.6 QUIC

QUIC is a connection-oriented protocol originally developed by Google. Nowadays it has evolved into an IETF Standard which diverges from the the Google proprietary solution (gQUIC). QUIC version 1 is described in RFC 9000 [33]. It is sometimes associated to the acronym Quick UDP Internet Connections. QUIC runs over UDP but provides reliable communication through the implementation of mechanisms such as flow control, congestion control, and loss detection. QUIC outdoes TCP in several aspects. First of

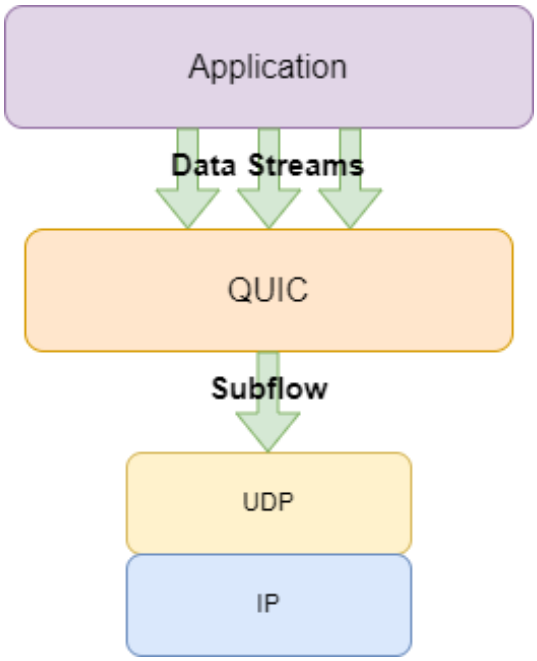


Figure 4.10: QUIC stack

all, the fact that TCP is implemented in kernel space makes the whole network infrastructure resistant to change, since a change in the TCP stack requires operating system modifications. QUIC is implemented in user space, which makes it easily upgradable and customizable. Secondly, unlike TCP, QUIC does not suffer from the Head of Line (HoL) problem, which occurs if there is a single queue of data packets waiting to be transmitted, and the packet at the head of the queue (line) cannot move forward due to congestion, even if other packets behind this one could. This is because a QUIC connection is made of streams that can be handled independently from one another, so that loss of packets of one does not have an impact on packets of the others. Thirdly, TCP is not secure by default and therefore almost all applications nowadays run TLS on top of it. QUIC includes instead TLS 1.3 handshake in its connection establishment process and is therefore an encrypted-by-default protocol. This leads to another advantage of QUIC that can be seen in figure 4.11, which is a lower connection time. While TCP with TLS on top of it requires three Round Trip Times (RTT) to establish a connection, a QUIC client needs only one RTT to establish a connection towards an unknown server and zero RTT to establish connections to known servers, meaning servers with which communication already happened at least once.

The connection establishment procedure is shown more in detail in figure 4.12 and works as follows. For the first time a client tries to establish a connection with a server, the client will send an incomplete client hello (CHLO) message to the server to receive a reject (REJ) message. The REJ message contains the server configuration, authentication certificate, signature for the server certificate, and source-address token. The source-address token

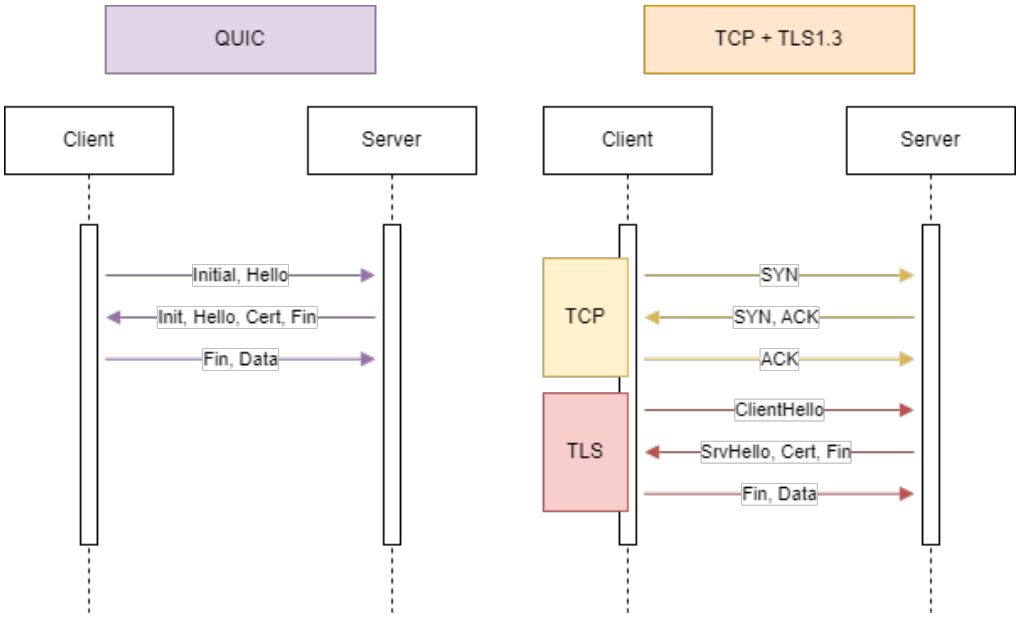


Figure 4.11: QUIC handshake vs TCP+TLS handshake

is used to verify the identity of the client in future communications. The client uses the information provided by the REJ message to construct the complete CHLO message. Then, the client sends a complete CHLO message that contains a temporary Diffie-Hellman public key for the client. Once the client has received the REJ message, it can start sending data without the initial incomplete CHLO and without waiting for a SHLO message from the server, thus achieving the 0-RTT handshake. After the client receives SHLO, the client starts sending data using final keys calculated from the information provided in the SHLO message.

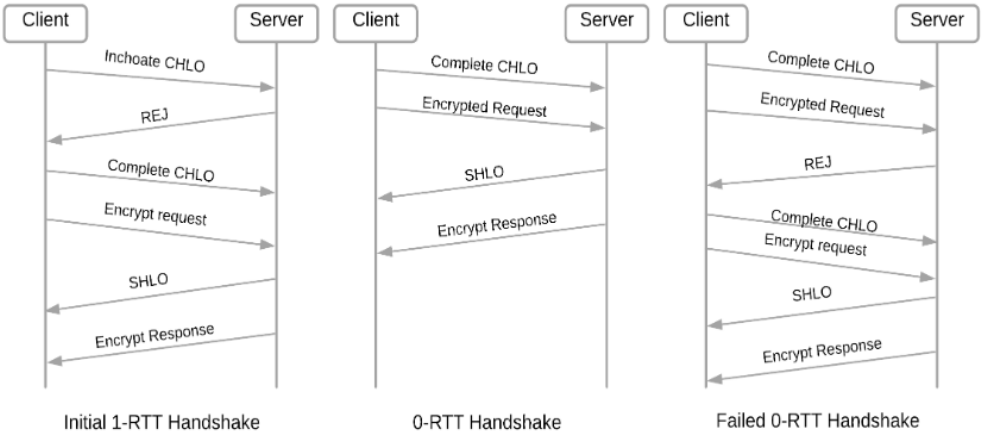


Figure 4.12: QUIC connection establishment. Reproduced from [34]

QUIC has pluggable congestion control, and provides richer information to the congestion control algorithm than TCP. Currently, Google's implementation of QUIC uses a reimplementation of TCP Cubic and is experimenting with alternative approaches. One example of such richer information is that each packet, both original and retransmitted, carries a new sequence number. This allows a QUIC sender to distinguish ACKs for retransmissions from ACKs for originals and avoids TCP's retransmission ambiguity problem. QUIC ACKs also explicitly carry the delay between the receipt of a packet and its acknowledgment being sent and, together with the monotonically-increasing sequence numbers, allows for precise RTT calculation. In conclusion, QUIC solves a number of transport-layer and application-layer problems experienced by modern web applications while requiring little to no modifications to their structure. One possible concern with QUIC is that the user space implementation may reduce the achievable performances of the protocol, as each message, including control messages, triggers a context switch between kernel and user spaces. Comparisons between TCP and an implementation of QUIC in the linux kernel [35] show that while they perform equally in lower loss rate measurements, QUIC generally outperforms TCP when packet loss is added.

MPQUIC

MPQUIC is a multipath extension to the QUIC protocol whose design, a prototype implementation in Go, and an initial performance evaluation have been presented quite recently in [36]. While regular QUIC already employs subflows to improve resiliency it can only use them as backup, meaning that nonprobing packets can only be exchanged along one path at a time. Path identification is achieved in MPQUIC by placing an explicit Path ID in the public header of each packet. This also allows the protocol to preserve states (congestion control, lost packets) for the paths even after an IP address change. MPQUIC uses a separate packet number space for each path, and also adds a Path ID to ACK frames. The protocol has a path manager component that is responsible for the creation and deletion of paths. Unlike MPTCP, MPQUIC is able to send data in the first packet as it opens a new path, while MPTCP requires a three-way handshake before any data is sent. MPQUIC also employs a more flexible retransmission mechanism compared to MPTCP, as lost packets can be retransmitted on different paths.

The authors of [37] show how the performance of an MPQUIC implementation is able to match or surpass that of MPTCP, highlighting however the need for further optimization. MPQUIC is still a very recent solution, but will likely see usage as regular QUIC becomes more widespread.

4.6.1 Integration with common protocols

QUIC was designed specifically to fit the requirements imposed by recent trends of web services. This is why it lends itself very easily to integration with the most popular protocols used today. We now describe two protocols that already have widespread use and that greatly benefit from usage of QUIC as transport layer.

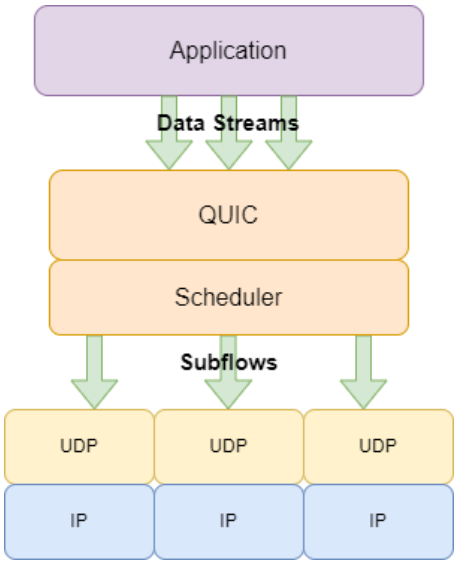


Figure 4.13: MPQUIC stack

HTTP3

In October 2018, the IETF HTTP and QUIC Working Groups jointly decided to refer to HTTP/3 as the HTTP mapping over QUIC, in advance of making it a worldwide standard. Adoption of this protocol is steadily rising, as it has been proven to perform better in bad network conditions [38].

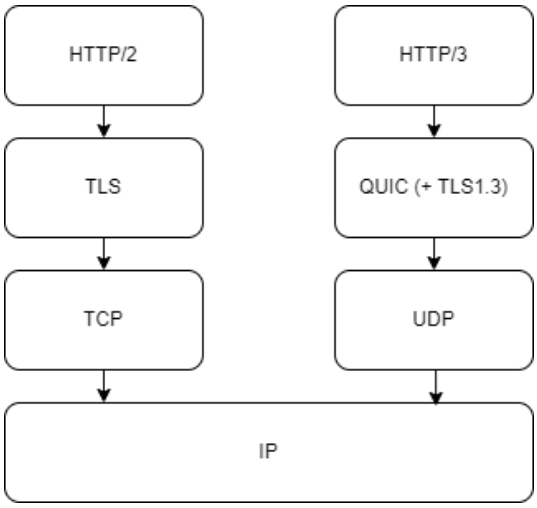


Figure 4.14: Comparison of HTTP/2 and HTTP/3 stacks

One of the larger issues with HTTP2 on top of TCP is the issue of head-of-line blocking. The application sees a TCP connection as a stream of bytes. When a TCP packet is lost, no streams on that HTTP2 connection can make forward progress until the packet

is retransmitted and received by the far side - not even when the packets with data for these streams have arrived and are waiting in a buffer. Because QUIC is designed from the ground up for multiplexed operation, lost packets carrying data for an individual stream generally only impact that specific stream. Each stream frame can be immediately dispatched to that stream on arrival, so streams without loss can continue to be reassembled and make forward progress in the application.

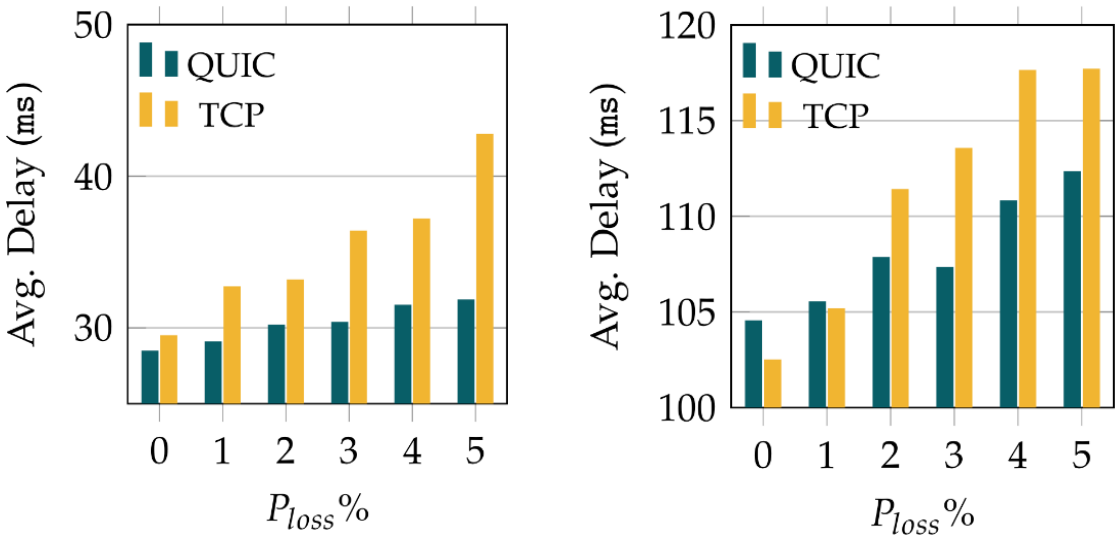
The HTTP3 client server logic is already available on several open source libraries which include `Proxygen`[39] and `Cronet`[40] for the C++ language (by Facebook and Google respectively), `nego`[41] for rust (by Mozilla), `aioquic`[42] for python and `quic-go`[43] for Go.

MQTT

MQTT is a popular application level protocol based on the publish subscribe paradigm. It has become extremely popular for IoT applications thanks to its ease of implementation, small code footprint, bandwidth efficiency, and client decoupling. The protocol defines three entities: publisher, subscriber, and broker. Connections are established between publisher and broker, and between subscriber and broker. Publishers act as clients and generate information, whose type is specified by a label called topic, and send it to the broker. Subscribers also act as clients, but they register at the broker topics they are interested in. Upon receiving messages from publishers, the broker filters them, according to their topic, and forwards them to the interested subscribers. MQTT is an application protocol, therefore two independent transport layer connections are necessary between each pair of entities. MQTT supports different QoS levels. There are three supported QoS modes:

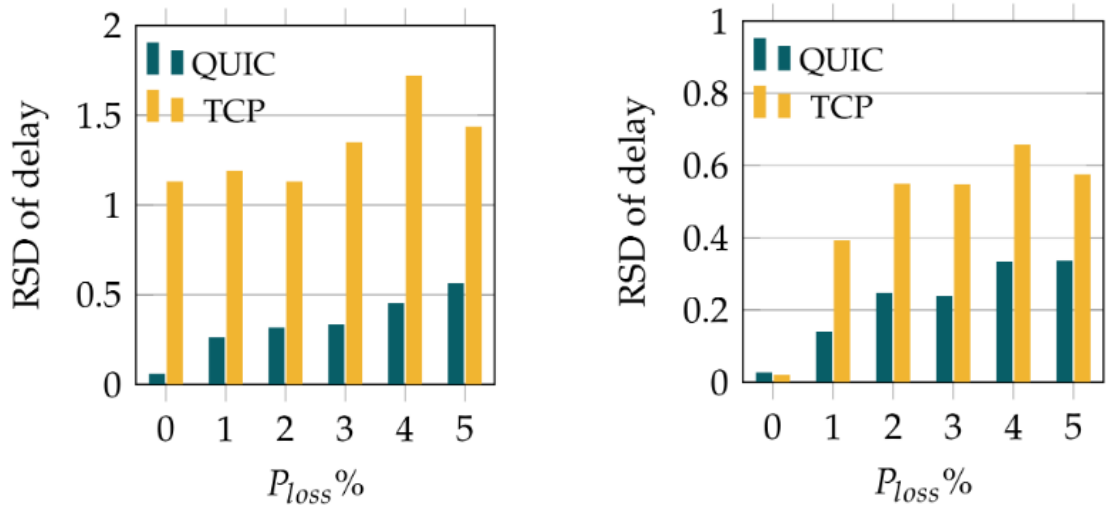
- QoS 0 (at most once): the message is sent only once, with no retries. Reliable communication can still be enforced by lower layers.
- QoS 1 (at least once): in this case, the message can be retransmitted until the sender receives an acknowledgment.
- QoS 2 (exactly once): the message is delivered exactly once. This is ensured with a four-way handshake to ensure that both the original message and its acknowledgment have been correctly received by receiver and sender, respectively. While it is the most reliable mode, it is also the slowest: it takes at least 2 RTTs to deliver a message.

Nevertheless, the MQTT protocol has inherent drawbacks in certain complex network environments due to underlying TCP transport protocol limitations. Currently there exist a few available MQTT libraries that integrate QUIC, one of which is EMQX. In their website is presented a comparison between QUIC and TCP as transport layer solutions [45]. They show that frequent connection interruptions due to network switching in TCP make it difficult to re-establish a connection after disconnection: the operating system is slow to release resources, the application layer cannot sense the disconnection status in



(a) WiFi (b) Cellular

Figure 4.15: Average delay observed for QUIC and TCP at different loss probabilities. Reproduced from [44].



(a) WiFi (b) Cellular

Figure 4.16: Relative Standard Deviation comparison for QUIC and TCP at different loss probabilities. Reproduced from [44].

time and the Server/Client overhead is high when reconnecting. In these scenarios, the low connection overhead and multi-path support of QUIC show their strengths. QUIC also opens up the possibility of connection multiplexing where independent topics could have independent streams within the same connection to eliminate interferences (such as

blocking from higher priority topics or flow control at the receiving side), and client/server side migration [46]. The server side migration discussed by the EMQX developers is aimed at preventing massive reconnections from the clients after migration within the same cluster, which means that it does not tackle our usecase in which the server's IP address changes.

According to the authors in [44], adoption of QUIC brings benefits on delay (25% on wifi links) and jitter (61% delay RSD Relative Standard Deviation reduction on wifi) at the price of slight energy consumption increase, with similar amounts of traffic generated, as shown in figures 4.15 and 4.16.

4.6.2 QUIC libraries

As mentioned in a previous section, there exist many libraries that implement the QUIC protocol. QUIC performance however strongly depends on the chosen implementation and open source libraries currently have very different performances from production environments like Google, Facebook and Cloudflare's [47]. The python library *aioquic* [42] was our implementation of choice because of the ease of use intrinsic to the programming language. It can be imported into a python script like any other library and features three APIs, described in this section.

QUIC API

The QUIC API performs no I/O on its own, leaving this to the API user. This allows you to integrate QUIC in any Python application, regardless of the concurrency model you are using. The most important class is *QuicConnection*, which acts as a state machine driven by the following kind of sources:

- the API user requesting data to be send out
- data being received from the network
- a timer firing

Initiating an instance of this class requires another object called *QuicConfiguration*, where all the connection parameters are defined (including whether the host is acting as a client or not).

HTTP/3 API

The HTTP/3 API performs no I/O on its own, leaving this to the API user. This allows you to integrate HTTP/3 in any Python application, regardless of the concurrency model you are using. The fundamental class here is *H3Connection*, which requires a *QuicConnection* object as parameter. Similarly to the QUIC API, data reception events are handled by event classes such as *DataReceived*.

asyncio API

The asyncio API provides a high level QUIC API built on top of asyncio, Python's standard asynchronous I/O framework. The *aioquic* library comes with an HTTP/3 server and client scripts already prepared based on this framework. To establish a server a host must run the function *serve*, which requires a *QuicConfiguration* containing TLS certificate and private key as arguments

4.6.3 QUIC applied to migration

A considerable difference between QUIC and TCP, which is much relevant to our work, is the way these protocols define and handle connections. Unlike TCP connections, which are uniquely identified by the 4-tuple <source IP, source port, destination IP, destination port>, QUIC connections are identified by a 64 bit connection ID, randomly generated by the client. After the connection is established, the communicating entities agree on a set of additional connection IDs to be used when the connection is migrated. This means that when a QUIC client changes IP addresses (changes the NIC through which data is sent, for example when moving outside of WiFi range and switching to cellular), it can continue to use the old connection from the new IP address without interrupting any in-flight requests by using a different connection ID from the set, as shown in figure 4.17. Therefore, if appropriate mechanisms are implemented, QUIC connections can be migrated to the new address (i.e. IP and port) of a migrating endpoint. As we have already discussed, this is not the case for TCP.

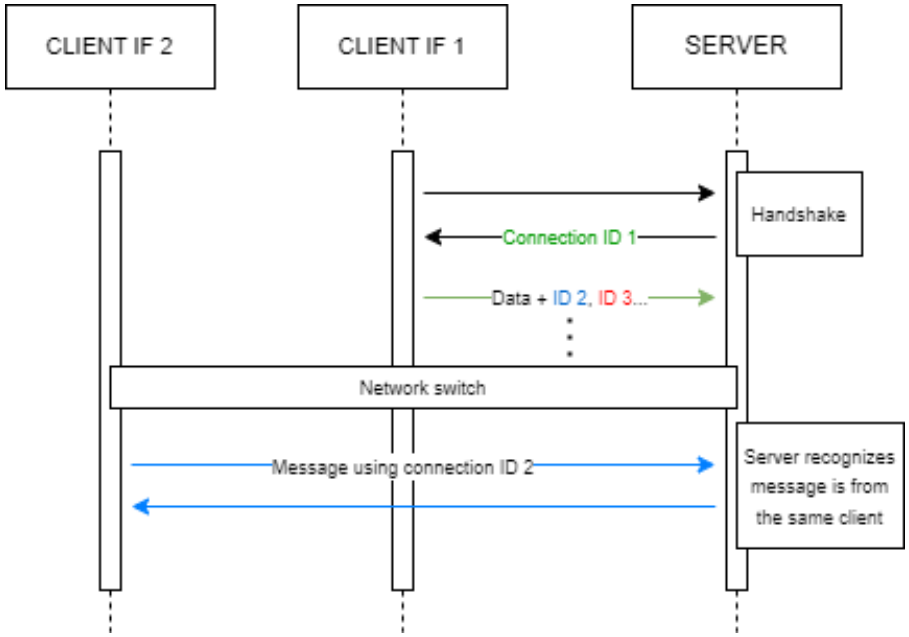


Figure 4.17: After the client switches to a different interface, it uses an ID from the set negotiated earlier allowing the server to keep the connection active.

Currently, only client-side connection migration is implemented in QUIC, and it works

as follows. After changing its address (handover), the client initiates the procedure by sending a non-probing packet to the server. The latter receives this packet and understands that the client has migrated to a new address. Hence, the server sets that address as the client's primary address (a.k.a. primary path) and starts a procedure called path validation, which verifies client reachability on the new address. The server validates the path by sending a PATH CHALLENGE frame to the client and waiting for a PATH RESPONSE frame. Path validation is successful when the PATH RESPONSE frame contains the same data that was sent in the corresponding PATH CHALLENGE frame. If path validation is successful, the active connection is migrated to the new address of the client. This validation procedure is performed only if the address had not been validated previously. The current version of QUIC does not allow the migration of a connection to a new server address mid-connection but only right after connection establishment. Namely, QUIC allows servers to accept connections on one IP address and then transfer these connections to a more preferred address shortly after the handshake. The procedure is the following: during connection establishment, a server can inform a client of a preferred address by including the preferred address transport parameter in the TLS handshake; after the handshake is confirmed, the client should start path validation towards the server preferred address; if path validation succeeds, the connection is migrated to the new server address.

Compared to the other presented protocols, it is clear that QUIC presents several traits that make it the most suitable for the implementation of the model described in 4.3. The appeal is not only the relative ease of implementation though, but also its popularity. As its usage increases, more and more servers/middleboxes will support it and implement optimizations, ensuring that research efforts on this protocol are not wasted. A very interesting solution has been proposed by the authors in [48], who design and implement two stateful migration solutions that we will analyze in detail.

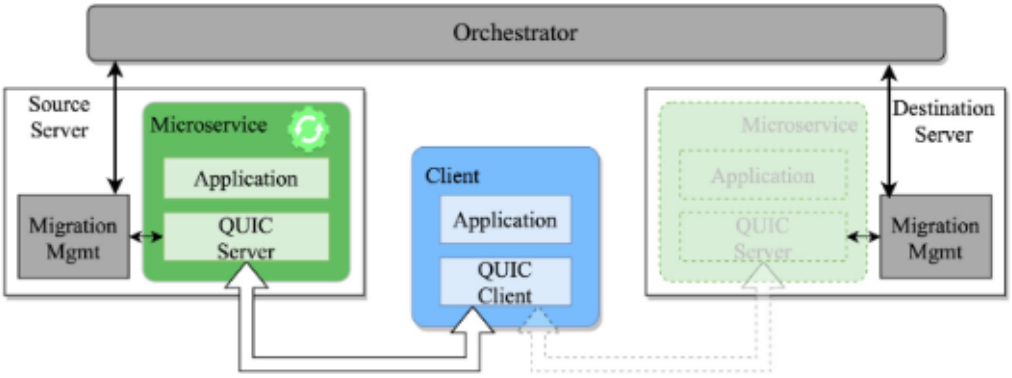


Figure 4.18: Stateful migration using QUIC. Reproduced from [48]

Figure 4.18 depicts their reference architecture. As shown, the end-user's device natively runs a client application and a client-side QUIC instance (i.e., QUIC client). The server machine, which could be an edge node or a cloud server, hosts two components. The first

one is a container that encapsulates a microservice, composed of a server application and a server-side QUIC instance (i.e., QUIC server). As a result, in this architecture, QUIC server is part of the container and is migrated along with it. The second component is instead indicated as migration management and represents a generic entity that handles container migration. This component acts as an intermediary between the container and an orchestrator, which may instruct it on when, where, and how (i.e., which technique) to migrate containers. In their work they propose three strategies to support server-side connection migration in QUIC (namely Reactive-Explicit, Proactive-Explicit and Pool-of-Addresses), which we will briefly report and discuss. They are all achieved by introducing minimal additions to both QUIC server and client in order to make them aware of migration, leveraging logic already present in the mechanism of client-side connection migration and the server's preferred address mechanism, thus preserving the normal flow of operations.

Explicit Strategies

Both the explicit strategies start with the QUIC server explicitly informed at runtime of an imminent container migration. The server immediately notifies the QUIC client with the exact destination address (i.e., IP and/or port) right before migration starts. This explicit information speeds up connection migration, as the destination address is deterministically known. The migration trigger received by the server can be either sent by the client through a new QUIC frame called TRIGGER frame (an option which the authors rightfully describe as useful in cases of user mobility) or through an API exposed to the migration management. In any case, the new IP address must be retrieved from the migration management. The QUIC server shares the necessary information to the client through a new QUIC frame called SERVER_MIGRATION frame. These frames can be encapsulated in standard QUIC packets and are therefore transmitted reliably.

The steps involved, described in figure 4.19, are the following:

1. QUIC server is triggered for migration

2. QUIC server sends a SERVER MIGRATION frame to QUIC client, advertising to the client the new address on which QUIC server is going to listen once migration is complete

3. QUIC client acknowledges the packet containing the SERVER MIGRATION frame to QUIC server on the old address.

4. container migration to the destination host can start. This is the step where the proactive and reactive strategies differ. The client does not know when the old server will become unreachable, so it either immediately assumes that the new address is active (proactive strategy) or it considers the old address active until a packet-loss is detected.

5. the QUIC client probes the new address of the server through the standard QUIC

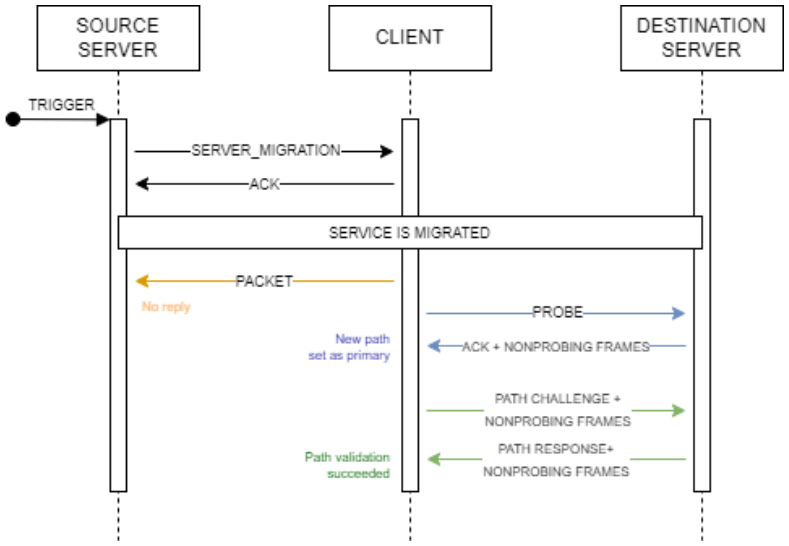


Figure 4.19: Sequence diagram for the explicit migration strategies. Sequence in yellow only happens in reactive explicit

mechanism that consists in sending a packet with a PING frame and possibly frames of the previously lost packet.

6. the QUIC server replies from the new address with a packet that can be either probing or non-probing, depending on what the QUIC client sent in the previous step. When QUIC client receives a non-probing packet from QUIC server, it sets the new path to the server as primary

7. if the new path to the server has not been validated before, QUIC client starts path validation as described by the QUIC specification

8. if the path is validated successfully, the QUIC connection is migrated to the new address of the server. Otherwise, in accordance with QUIC specification, QUIC client tries to reach the server on the previously validated paths, sending packets to those addresses. If QUIC server does not respond, the connection is closed.

Pool of Addresses

The Pool-of-Addresses (PoA) strategy assumes that the container can be migrated within a known in advance set of server machines. During connection establishment, QUIC server notifies QUIC client with the pool of possible destination addresses. This pool of destination addresses can be either provided as a parameter when a container is launched or configured in the container base image. The first case is targeted towards flexibility, as each container may be provided with a different pool of addresses, while the second case is targeted towards convenience, as all containers spawned from that image will share the same pool of addresses without additional configuration. This solution fits those cases in which the IP address of the target container is not known deterministically, and at

the same time removes the need for any interaction with the migration management at runtime. The QUIC client is notified of the pool of possible addresses through several POOL MIGRATION ADDRESS frames, a new type of frame designed by the authors. In any moment, the container can be migrated to another server machine without QUIC server or QUIC client being previously informed of that. When the QUIC client is unable to reach the QUIC server on its current primary address, it probes addresses from the pool to find where QUIC server has migrated.

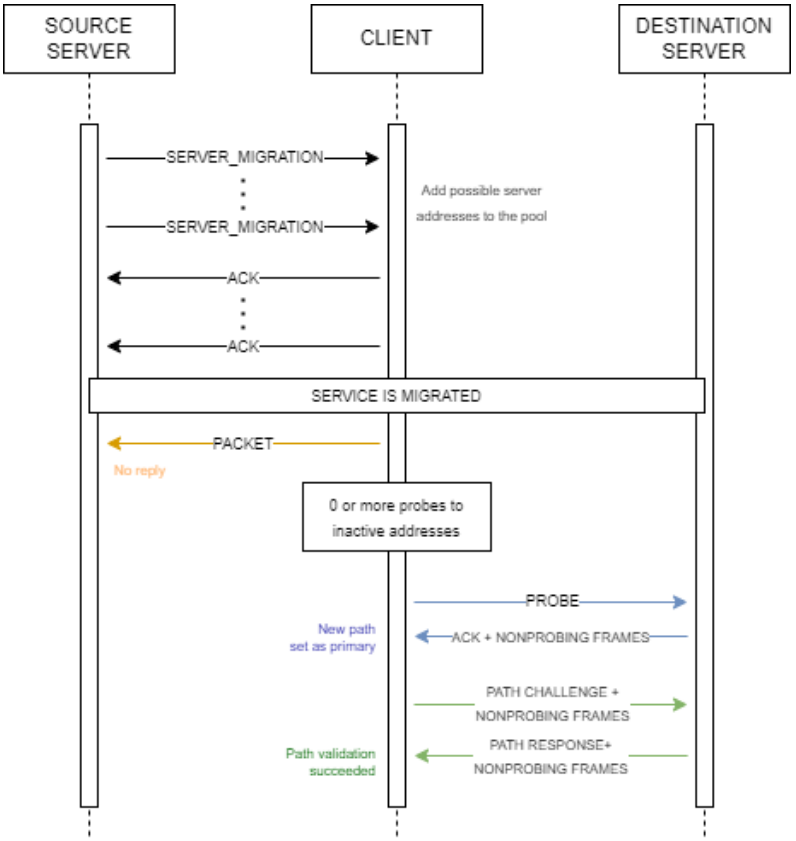


Figure 4.20: Sequence diagram for the PoA migration strategy

This strategy, described in figure 4.20, consists of the following steps:

1. During the initial handshake, for each address inside the pool, the QUIC server sends a POOL_MIGRATION_ADDRESS frame to QUIC client. These frames are acknowledged and no further action is required until migration happens

2. the QUIC server migrates to one of the addresses inside the pool

3. after each packet loss, the QUIC client considers the possibility that QUIC server has migrated and probes the addresses from the pool, including the current primary address in case the missing reply is simply due to congestion. The probing mechanism is the same as with the explicit strategies, sending a packet with a PING

frame and possibly frames of the previously lost packet.

4. the remaining steps are the same as the ones in the explicit strategy

Comments

The authors test their solution in two scenarios: in case of sporadic requests by the client and continuous back-to-back requests. As predictable the explicit strategies perform better that the PoA and should therefore be preferred in any of those cases in which a central entity able to orchestrate the container-migration procedure is available, meaning an entity able to dynamically know the new container IP address and inform the QUIC server. On the other hand, the PoA solution is suitable for those cases in which there is no central control on the container migration procedure. One great advantage of this proposed solution is that only the QUIC logic is subject of changes, so if QUIC is provided as a library the extension can be integrated with no impact on either the client or the server application.

Chapter 5

Conclusion

The developments in communication technologies have seen a surge in the popularity of IoT applications. Moving the computing resources closer to the edge of the network, called edge computing paradigm or MEC, has been identified as the solution to achieve scalability of network resources and acceptable communication delays. Edge servers however come with the tradeoff of reduced coverage and limited computing resources, which makes service migration fundamental to preserve the reduced latency. Containers are a lightweight virtualization technology that has been identified as the standard in MEC, therefore the study of migration in edge scenarios can be generalized with the study of container migration. Meanwhile, multipath transport protocols are receiving increasingly widespread attention in both literature and production environments. Their ability to create subflows from several NIC is particularly useful in edge scenarios, where it can be used to improve reliability and bandwidth. In this work we have discussed the role of edge computing, with a focus on the necessities of mobile users, and why stateful migration is relevant to meet the performance requirements of mobile users. We have given an overview of virtualization technologies, following with a detailed explanation of the state of the art techniques used for stateful migration and the tools commonly used during the migration process. Focusing on the problems related to the change in IP address of the migrated services, we identified multipath protocols as a possible solution to avoid reconnection downtime associated with classic TCP connections. Through a focused literature review, we analyzed the features of three multipath protocols, MPTCP, MPDCCP and (MP)QUIC, regarding their application to the envisioned stateful migration scenario. Our findings on each of them is summarized in the following.

MPTCP

MPTCP is a multipath protocol that employs TCP in each of its subflows and is widely researched in the literature. It's available in all linux kernel stacks from version 5.6 onward and maintained by the community. Several solutions have been proposed to address stateful migration, however the nature of the protocol requires modifications to the kernel in order to implement the proposed changes. This hurts the deployability of

the analyzed solutions.

MPDCCP

MPDCCP is a protocol that is heavily specialized for real time media traffic. While this is useful in some scenarios, it is still sees no widespread adoption and literature on the topic is very limited. No published papers have considered its adoption to support migration of stateful container applications, most likely because it shares the same disadvantages as MPTCP in that it requires modifications to the kernel stack but does not bring any noticeable benefit. For these reasons it is not deemed suitable to enable stateful container migration.

QUIC

QUIC is a recently standardized protocol designed to run in user space. It leverages UDP at the transport layer, but implements congestion control and retransmission mechanisms at a higher layer. It is designed as a secure-by-default protocol, integrating the TLS handshake into its own and thus reducing the overhead during the startup. As we have seen QUIC's future is looking bright thanks to the growing adoption rate of HTTP/3. Moreover integration with MQTT is starting to become commercially available on open source libraries, increasing the potential future use cases. As these two protocols are the most widely used in IoT applications, QUIC is sure to be widely employed and is therefore a good candidate for research. We have seen how the client IP address migration procedure already featured by QUIC can be easily extended to allow for server migration.

5.1 Future Work

The application of multipath protocols to stateful migration is a topic that is still in its early stages of research. Most of the proposed solutions focus on a purely isolated environment and do not take into account the effects of middleboxes or the integration with edge network controllers. It has been shown that both QUIC and MPTCP can interact in unfavorable ways with middleboxes that do not accept UDP packets or that discard TCP packets with unknown options respectively. Further testing of these solutions in the wild is necessary to raise awareness of such interactions. Another aspect that requires further research is the integration of the currently existing solutions with entities capable of recognizing the need of migration and trigger the procedure providing the destination IP address, such as network controllers and/or orchestrators. Finally, we have discussed how QUIC implementations vary in performance. Therefore to accurately gauge field performance, the presented server migration procedure in QUIC should be implemented and tested with more focus on performance optimization.

Bibliography

[1] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[2] Meikang Qiu. Container memory live migration in wide area network. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 12608 of *Lecture Notes in Computer Science*, pages 68–78. Springer International Publishing AG, Switzerland, 2021.

[3] runc. `https://github.com/opencontainers/runc`. Accessed: 2023-10.

[4] Criu: Checkpoint restore in userspace. `https://criu.org/Main_Page`. Accessed: 2023-10.

[5] Docker. `https://www.docker.com/`. Accessed: 2023-10.

[6] Podman. `https://podman.io/`. Accessed: 2023-10.

[7] Shahidullah Kaiser, Ali Saman Tosun, and Turgay Korkmaz. Benchmarking container technologies on arm-based edge devices. *IEEE access*, 11:107331–107347, 2023.

[8] Singularity. `https://sylabs.io`. Accessed: 2023-10.

[9] Borislav Dordevic, Valentina Timcenko, Milovan Lazic, and Nikola Davidovic. Performance comparison of docker and podman container-based virtualization. In *2022 21st International Symposium INFOTEH-JAHORINA (INFOTEH)*, pages 1–6, Piscataway, 2022. IEEE.

[10] Enzo Baccarelli, Michele Scarpiniti, and Alireza Momenzadeh. Fog-supported delay-constrained energy-saving live migration of vms over multipath tcp/ip 5g connections. *IEEE access*, 6:42327–42354, 2018.

[11] M.2083. Imt vision - framework and overall objectives of the future development of imt for 2020 and beyond, 2015.

[12] v16.4 Standard 23.501. System architecture for the 5g system, 2020.

[13] Yenchia Yu, Antonio Calagna, Paolo Giaccone, and Carla Fabiana Chiasserini. Tcp connection management for stateful container migration at the network edge. In *2023 21st Mediterranean Communication and Computer Networking Conference (MedComNet)*, pages 151–157. IEEE, 2023.

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

[15] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath tcp. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 399–412, San Jose, CA, 2012. USENIX.

[16] Ramin Khalili, Nicolas Gast, Miroslav Popovic, and Jean-Yves Le Boudec. Mptcp is not pareto-optimal: Performance issues and a possible solution. *IEEE/ACM transactions on networking*, 21(5):1651–1665, 2013.

[17] Per Hurtig, Karl-Johan Grinnemo, Anna Brunstrom, Simone Ferlin, Ozgu Alay, and Nicolas Kuhn. Low-latency scheduling in mptcp. *IEEE/ACM transactions on networking*, 27(1):302–315, 2019.

[18] Kadiyala Ramana, Rajanikanth Aluvalu, Vinit Kumar Gunjan, Ninni Singh, and M. Nageswara Prasadhu. Multipath transmission control protocol for live virtual machine migration in the cloud environment. *Wireless communications and mobile computing*, 2022:1–14, 2022.

[19] Jinsung Lee, Youngbin Im, and Joohyung Lee. Modeling mptcp performance. *IEEE communications letters*, 23(4):616–619, 2019.

[20] Rajnish Kumar Chaturvedi and Satish Chand. Multipath tcp security over different attacks. *Transactions on emerging telecommunications technologies*, 31(9), 2020.

[21] et al. C. Paasch, S. Barre. Multipath tcp in the linux kernel. `http://www.multipath-tcp.org`.

[22] Community maintained mptcp in linux kernel. `https://www.mptcp.dev`.

[23] mptcpd. `https://github.com/multipath-tcp/mptcpd`.

[24] Yuqing Qiu, Chung-Horng Lung, Samuel Ajila, and Pradeep Srivastava. Experimental evaluation of lxc container migration for cloudlets using multipath tcp. *Computer networks (Amsterdam, Netherlands : 1999)*, 164:106900–, 2019.

[25] Nitinder Mohan, Tanya Shreedhar, Aleksandr Zavodavoski, Otto Waltari, Jussi Kangasharju, and Sanjit Kaul. Redesigning mptcp for edge clouds. In *Proceedings of the 24th Annual International Conference on mobile computing and networking*, MobiCom '18, pages 675–677. ACM, 2018.

[26] Zhao Haitao, Ding Yi, Zhang Mengkang, Wang Qin, Shi Xinyue, and Zhu Hongbo. Multipath transmission workload balancing optimization scheme based on mobile edge computing in vehicular heterogeneous network. *IEEE Access*, 7:116047–116055, 2019.

[27] Catalin Nicutar, Christoph Paasch, Marcelo Bagnulo, and Costin Raiciu. Evolving the internet with connection acrobatics. In *Proceedings of the 2013 workshop on hot topics in middleboxes and network function virtualization*, HotMiddlebox '13, pages 7–12. ACM, 2013.

[28] Franck Le and Erich M. Nahum. Experiences implementing live vm migration over the wan with multi-path tcp. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1090–1098. IEEE, 2019.

[29] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald SchiÃ¶berg. Live wide-area migration of virtual machines including local persistent state. In *ACM/Usenix International Conference On Virtual Execution Environments: Proceedings of the 3rd international conference on Virtual execution environments; 13-15 June 2007*, VEE '07, pages 169–179. ACM, 2007.

[30] Sally Floyd, Mark J. Handley, and Eddie Kohler. Datagram Congestion Control Protocol (DCCP). RFC 4340, March 2006.

[31] Sally Floyd and Eddie Kohler. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control. RFC 4341, March 2006.

[32] Jitendra Padhye, Sally Floyd, and Eddie Kohler. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC). RFC 4342, April 2006.

[33] M. Thomson J. Iyengar. Quic: A udp-based multiplexed and secure transport, 2021.

[34] Saleh Alawaji. Ietf quic v1 design, 2021.

[35] Peng Wang, Carmine Bianco, Janne RiihijÃ¤rvi, and Marina Petrova. Implementation and performance evaluation of the quic protocol in linux kernel. In *Proceedings of the 21st ACM International Conference on modeling, analysis and simulation of wireless and mobile systems*, MSWIM '18, pages 227–234. ACM, 2018.

[36] Quentin De Coninck and Olivier Bonaventure. Multipath quic: Design and evaluation. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, page 160â166, New York, NY, USA, 2017. Association for Computing Machinery.

[37] Zsolt KrÃ¤mer, FeliciÃ¡n NÃ©meth, Attila MihÃ¡ly, SÃ¡ndor MolnÃ¡r, IstvÃ¡n Pelle, Gergely PongrÃ¡cz, and DonÃ¡t Scharnitzky. On the potential of mp-quic as transport layer aggregator for multiple cellular networks. *Electronics (Basel)*, 11(9):1492, 2022.

[38] Martino Trevisan, Danilo Giordano, Idilio Drago, and Ali Safari Khatouni. Measuring http/3: Adoption and performance. IEEE, 2021.

[39] Facebook. proxygen. https://github.com/facebook/proxygen#quic-and-http3. Accessed: 2023-09.

[40] Google. Cronet. https://github.com/chromium/chromium/tree/master/net/quic. Accessed: 2023-09.

[41] Mozilla. neqo. https://github.com/mozilla/neqo. Accessed: 2023-09.

[42] aiortc. aioquic. https://github.com/aiortc/aioquic. Accessed: 2023-09.

[43] Marten Seemann Lucas Clemente. quic-go. https://github.com/lucas-clemente/quic-go. Accessed: 2023-09.

[44] Sidna Jeddou, FÃ¡tima FernÃ¡ndez, Luis Diez, Amine Baina, Najid Abdallah, and RamÃ³nAgÃijero. $Delayandenergyconsumptionofmqttoverquic : Anempiricalcharacterizationusingcommercial - off - the - shelfdevices. Sensors(Basel, Switzerland), 22(10) : 3694 - -, 2022.$

[45] EMQX Team. Mqtt over quic: Next-generation iot standard protocol. https://www.emqx.com/en/blog/mqtt-over-quic#quic-vs-tcp-tls. Accessed: 2023-09.

[46] William Yang. Mqtt over quic. https://www.oasis-open.org/committees/download.php/69611/oasis_mqtt_over_quic.pdf, 2022.

[47] Alexander Yu and Theophilus A. Benson. Dissecting performance of production quic. In *Proceedings of the Web Conference 2021*, WWW '21, page 1157â1168, New York, NY, USA, 2021. Association for Computing Machinery.

[48] Carlo Puliafito, Luca Conforti, Antonio Virdis, and Enzo Mingozzi. Server-side quic connection migration to support microservice deployment at the edge. *Pervasive and mobile computing*, 83:101580, 2022.