Server-side QUIC Connection Migration to Support Microservice Deployment at the Edge

Carlo Puliafito[a,*], Luca Conforti[a], Antonio Virdis[a], Enzo Mingozzi[a]

[a]*Department of Information Engineering, University of Pisa, Largo Lucio Lazzarino 1, 56122, Pisa, Italy*

Abstract

In edge computing environments, microservices are typically deployed in the form of containers. To maintain proximity between edge computing services and mobile users, containers need to be migrated between nodes. When migrating containers, however, it is important to consider that they typically have ongoing communications with client endpoints. Moreover, in case of connection-oriented protocols, communicating endpoints share a state (i.e., the connection), which needs to be migrated as well. Connection-oriented protocols like TCP were not designed having connection migration in mind, thus their connections cannot survive a change of IP address or port number. On the other hand, QUIC, a transport protocol recently standardised by IETF, provides a mechanism for client-side connection migration, whereas a server-side connection migration is not yet implemented nor investigated. In this work, we propose an extension of QUIC to support server-side connection migration when a container is migrated between servers. We designed three different strategies, fitting a diverse set of scenarios, wherein the migration procedure is either or not supported by a centralised entity, e.g., an orchestrator. We implemented and verified the proposed extension. Besides, we evaluated it on a real testbed, showing how each of the three strategies is impacted by different container migration techniques and container sizes. To conclude, we compared our solution against two alternatives based on TCP+DNS and MPTCP respectively, demonstrating performance improvements in terms service-migration time.

Keywords: QUIC, Edge computing, Mobility, Container migration, Service continuity, Internet of Things

*Corresponding author

Email address: carlo.puliafito@ing.unipi.it (Carlo Puliafito)

1. Introduction

Microservices are rapidly gaining momentum as a powerful architectural style to deliver software services to end users. Following this pattern, applications are designed and developed as a collection of one or more modules – microservices – that are loosely coupled and can be deployed and scaled independently [1]. Nowadays, microservices are mostly packaged and deployed using *container* frameworks, as opposed to Virtual Machines (VMs). In fact, containers are lightweight, faster to boot up, and quicker to migrate compared to VMs, mainly because they do not need their own entire OS. Moreover, the cost of deployment significantly reduces thanks to the ability to achieve a high utilisation via down-/up-scaling of container instances on the provider side [2].

At the same time, *edge computing* is powerfully emerging as an extension of cloud computing towards the network edge. With edge computing, services are pulled from large but remote data centres towards a pervasive infrastructure of geo-distributed micro data centres that are deployed close to (or co-located with) access networks. This proximity enables low latency, high bandwidth, and improved privacy, which are of utmost importance to demanding applications such as Augmented/Virtual Reality, autonomous vehicles, and the Internet of Things [3]. As such, edge computing is actually one of the driving forces pushing the design and deployment of future 5G/6G *mobile* access networks and services [4].

Delivering edge-hosted microservices is highly attractive, since it allows to reap the benefits of both, but also opens several challenges. In fact, mobile end users at the edge may roam across different network access points. When this happens, microservices associated to the user may need be migrated among edge servers in different micro data centres to remain close to the client [5, 6]. This can be done by performing *stateful* migration of the containers hosting the microservices. Stateful migration transfers the state of the container (i.e., memory pages, CPU state, disk, etc.) from the source to the destination server so as to preserve its execution state [7, 8].

However, container migration across different (micro) data centres typically results in a change of the IP address of the *server-side endpoint* of the communication between the user client and the associated microservice. To guarantee service continuity, a mechanism is hence needed to hold or re-establish the connection between them. Existing solutions leverage network virtualisation techniques or introduce ad-hoc protocols complemented by dedicated forwarding functions deployed

in the network [9, 10, 11]. Alternatively, they lean on the client application to discover the new location of the service (e.g., by dynamic DNS) and re-establish the (TCP-based) connection, while supporting continuity with the cooperation of the source server to buffer and redirect packets towards the destination server during the migration [12, 13]. As such, they generate additional overhead, which may also affect the performance of the service, and add complexity to the management of the network as well as to application development, if not done transparently to higher layers.

To avoid the overhead and complexity of existing approaches, we propose in this work a QUIC-based solution to support server-side container migration. QUIC is a connection-oriented protocol originally developed by Google and now evolved into an Internet Standard [14]. 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 [15]. Today, QUIC is supported by all major browsers and has been adopted to deliver many business web services, with fastly increasing use by mobile users [16].

QUIC presents several strengths over TCP, one being the capability to support client-side connection migration: when a client changes its IP address (e.g., after a network handover), QUIC transparently manages the migration of existing connections to the new IP address without disrupting ongoing transfers. On the other hand, server-side connection migration in QUIC is not yet specified nor investigated, even though it has been foreseen by the QUIC Working Group [17]. In this work, we extend QUIC to support server-side connection migration when a container hosting a QUIC-based service is migrated from a source to a destination server across different data centres[1]. Our solution, together with the already available client-side connection migration mechanism, allows QUIC to seamlessly handle a change of IP address by any endpoint involved in the connection. More specifically, this work makes the following contributions overall:

- It extends QUIC to support server-side connection migration through three different strategies, namely, Reactive-Explicit, Proactive-Explicit, and Pool-of-Addresses.
- It evaluates the performance of our solution through a set of experiments carried out over a

[1] A preliminary version of this work has been presented at IEEE WoWMoM 2021 [18], where we designed and implemented two strategies of server-side connection migration in QUIC, tested interoperability with other QUIC implementations, and carried out an initial performance evaluation of our solution.

realistic edge computing testbed, considering various metrics and covering different scenarios.

- It presents a quantitative comparison with a solution based on TCP+DNS, and a qualitative one with a solution based on MPTCP.

The remainder of the paper is organised as follows. Section 2 presents the background concepts on stateful container migration and QUIC, whereas Section 3 describes the design of the three strategies that we propose for server-side connection migration in QUIC. In Section 4, we evaluate our solution in two configuration scenarios. Finally, Section 5 discusses the state of the art on connection migration, and in Section 6 we draw the conclusions and outline the future work.

2. Background

This section provides an overview of the core concepts that serve as a basis for this work. We first discuss stateful container migration. Then, we provide information about QUIC, with a specific focus on its connection definition and handling.

2.1. Stateful container migration

The state of a container is composed of two main parts. One part is the container image, which includes the guest OS file system and an idle version of the application together with any application-specific data. The other part is the in-memory state, which consists of all the resources that are currently loaded in memory for quick access (e.g., memory pages, CPU state, content of registers) and represents the progress of a running application. Migrating the whole state of a container is often impractical, especially for demanding edge applications, as it causes clients to experience long service interruptions. Aiming at optimising the migration process, in our work we instead leverage the approach described in [19]. This approach exploits the fact that the container image can be proactively distributed across the edge servers, for example based on demand prediction. As a result, migrating a running container involves transferring only its in-memory state, thus significantly improving migration performance. The four most popular techniques to migrate the in-memory state of a container are described in the following.

Cold migration: it (i) first stops container execution to ensure that state is no longer modified; (ii) then checkpoints the state and transfers it while the container is stopped; and (iii) finally resumes

container execution at destination. As a result, cold migration may cause long container downtimes (i.e., the time interval during which the container is not running). Downtime even coincides with the migration time, which is the overall time required to migrate container state.

Pre-copy migration: in the initial phase (i.e., pre-dump), pre-copy migration checkpoints and transfers the container state while the container is running. Then, in the dump phase, it stops the container and transfers only the modifications to the state that occurred during the pre-dump phase. Finally, it restores container execution at destination. Downtime is typically shorter than that of cold migration, since only modifications to the state are transferred. However, migration time is longer because part of the state (i.e., that being modified) is copied more than once.

Post-copy migration: it first suspends container execution at source and copies a minimal part of the state (i.e., CPU state, registers) to the destination, allowing container execution to resume there. Then, while the container is running at destination, this technique transfers the rest of the state (i.e., faulted pages), which is most of it, in the background. During this last phase, container execution presents degraded performance because a considerable part of the state is missing. However, downtime for post-copy migration is very short.

Hybrid migration: it is a combination of pre-copy and post-copy. The pre-dump phase is the same as for pre-copy migration. The container is then stopped and only the modifications to CPU state and registers are copied to the destination. Finally, the container resumes its execution at destination, and, while it is running, the remaining part of the state (i.e., faulted pages) that was modified during the pre-dump phase is copied to the destination. In general, hybrid migration presents the shortest downtime, but shares the same shortcomings with pre-copy and post-copy.

Further details on these techniques as well as a performance comparison are available in [7]. At the time of writing, CRIU [20] is the basic tool to implement all the above techniques. CRIU allows to checkpoint in-memory state as a collection of files on disk, and to restore the container from that checkpoint in a later moment. CRIU needs to be complemented with a file transfer mechanism, such as `rsync`, to copy the state from the source to the destination server.

2.2. The QUIC Protocol

QUIC is a connection-oriented transport protocol which has been recently standardised by IETF under RFC 9000 [14]. It runs over UDP but provides reliable communication through the imple-

mentation of mechanisms such as flow control, congestion control, and loss detection. QUIC outdoes TCP in several aspects. Firstly, TCP runs in kernel space, which means that pushing changes to TCP stacks typically requires operating system upgrades. On the other hand, QUIC is implemented in user space. Secondly, QUIC avoids the head-of-line blocking problem that afflicts TCP. Namely, streams within a QUIC connection can be handled independently from one another, so that loss of packets of a stream does not have an impact on packets of other streams. Thirdly, TCP is not secure by default and hence needs TLS to run over it as an additional protocol. QUIC, instead, is an encrypted-by-default protocol that already includes TLS 1.3 handshake in its connection establishment process. This leads to a fourth advantage of QUIC, i.e., less time to establish connections. TCP with TLS on top of it, in fact, requires three Round Trip Times (RTT) to establish a connection. Differently, a QUIC client needs one RTT to establish a connection towards an unknown server, while connections to known servers are established with zero RTT.

Another considerable difference between QUIC and TCP, which is much relevant to our work, is the way these protocols define and handle connections. TCP connections are uniquely identified by the 4-tuple <source IP, source port, destination IP, destination port>. When any of these elements changes, the TCP connection is closed, and a new one must be established. QUIC, instead, defines a set of connection identifiers for each connection, which are independent from network parameters such as IP address or port. Therefore, if appropriate mechanisms are implemented, QUIC connections can be migrated to the new address (i.e., IP and port) of a migrating endpoint.

Currently, only client-side connection migration is implemented in QUIC, and it works as follows. After changing its address (e.g., due to a wireless 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. Path validation consists in the server 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 were sent in the corresponding PATH CHALLENGE frame. If path validation is successful, the active connection is migrated to the new address of the client. We highlight that validation is performed only if the address has not been validated previously.

Finally, we make a clarification with respect to server-side connection migration. Migrating a connection to a new server address mid-connection is not possible using the current version of QUIC. However, QUIC allows to migrate a connection to a new server address 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. 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. In the following section, we describe the design of our solution to migrate server-side connections in QUIC mid-connection.

3. QUIC Extension Design

This section describes the design of our solution by presenting a reference architecture and discussing three strategies to support server-side connection migration in QUIC.

3.1. Reference architecture and general design

Fig. 1 depicts our 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 - e.g., an edge node or a cloud server - hosts two components. The first one is indicated as migration management and represents a generic entity that handles container migration. This component can interact with an orchestrator, which may instruct it on when, where, and how (i.e., which technique) to migrate containers. This is however out of the scope of this work. The second component is instead a container that encapsulates a microservice, composed of a server application and a server-side

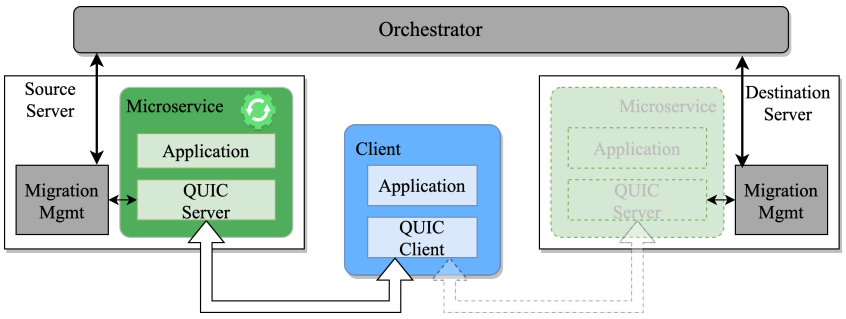


Fig. 1: Reference architecture.

QUIC instance (i.e., QUIC server). As a result, in our architecture, QUIC server is part of the container and is migrated along with it. We consider the possibility for QUIC server to interact with the migration management by means of an API that is exposed by QUIC server and is consumed by the migration management according to a protocol such as SNMP. This interaction is actually leveraged by the two Explicit strategies that we propose in this work to migrate server-side QUIC connections. The servers executing the microservice before and after migration are referred to as source and destination server respectively.

In this work, we propose three strategies to support server-side connection migration in QUIC (i.e., Reactive-Explicit, Proactive-Explicit and Pool-of-Addresses). All these strategies are based on the idea that server-side connection migration can be achieved by introducing minimal and non-breaking additions to both QUIC client and server, which are hence made aware of migration. These additions take inspiration from the mechanism of client-side connection migration as well as from the mechanism of the server's preferred address, which QUIC already defines (see 2.2).

The three strategies differ in the following respects. With both the Explicit strategies, QUIC server is explicitly informed at runtime of an imminent container migration and notifies 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. Reactive-Explicit and Proactive-Explicit then differ from one another for the moment in which they start sending packets to the destination address, as it is explained in Section 3.2. On the other hand, 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. With this strategy, QUIC does not deterministically know the next destination address, which should worsen overall performance. However, the PoA strategy, unlike the Explicit ones, allows to pre-provision the pool of destination addresses, without the need for any interaction with the migration management at runtime.

As we will see in the performance evaluation, the two Explicit strategies, on the one hand, and the PoA solution, on the other, cannot be considered as two equivalent alternatives, having the former two performing better than the latter. However, there are scenarios wherein the Explicit strategies would not be feasibly applied, whereas the PoA could. As a matter of fact, the Explicit strategies should be preferred in any of those cases wherein there is a central entity orchestrating

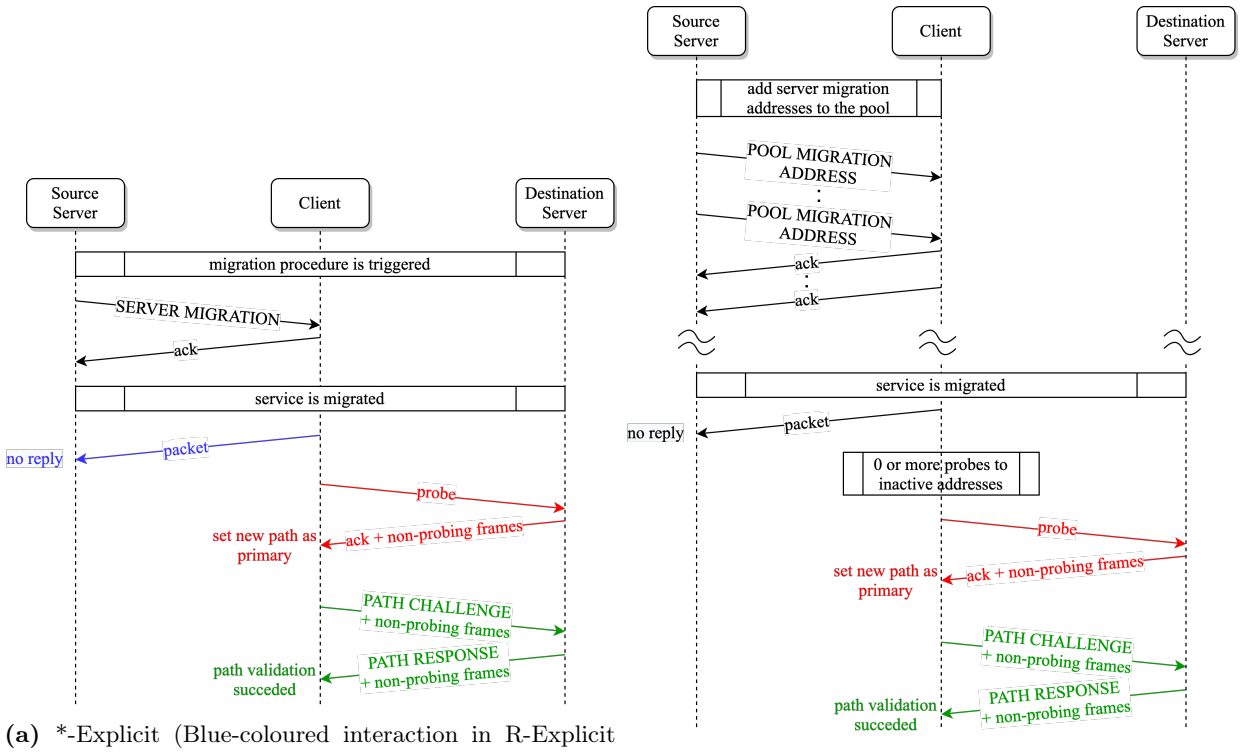
the container-migration procedure. Said *orchestrator* would know dynamically the new IP address of the server, and could interact with the QUIC server. On the other hand, the PoA solution is suitable for those cases wherein there is no central control on the container-migration procedure.

Although our extension requires modifications to both ends of the communication, i.e., the QUIC client and server, they only involve the QUIC logic, leaving the application logic untouched and ensuring an execution that is migration agnostic. As an example, if QUIC is provided as a library, our extension can be integrated with no impact on either the client or the server application. In what follows, we discuss the distinctive features of each of the three proposed strategies.

3.2. Reactive and Proactive Explicit

With these strategies, QUIC server is explicitly informed that container migration is going to start soon. This information can be conveyed in either of two ways. The first way is such that the migration management consumes the API exposed by QUIC server to let the latter know both the imminent beginning of container migration and the destination address. Alternatively, QUIC client can notify QUIC server of an impending migration by sending a new QUIC frame called TRIGGER frame. This second approach may be useful in mobility use cases in which it is the mobile device to trigger container migration. However, it is worth noting that, also in this case, QUIC server retrieves the destination address by interacting with the migration management. Whichever is the approach used, QUIC server must inform QUIC client of this imminent migration and communicate the new address on which the server is going to be listening to. To this purpose, we define a new QUIC frame called SERVER MIGRATION frame. Container migration cannot begin before QUIC server receives an acknowledgement to that frame from QUIC client. We highlight that the normal flow of operations in QUIC is preserved by our strategies. In fact, other non-probing frames (e.g., STREAM frames) can be encapsulated in QUIC packets together with TRIGGER and SERVER MIGRATION frames or their acknowledgements. Finally, it is worth noting that all the newly introduced frames are encapsulated into QUIC packets, which means that they are exchanged reliably through the QUIC connection. As an example, in case the server does not receive an acknowledgement to a SERVER MIGRATION frame, a retransmission will be performed.

Fig. 2a reports the sequence diagram of a successful connection migration according to the Explicit strategies. The steps involved are the following:



(a) *-Explicit (Blue-coloured interaction in R-Explicit only)

(b) Pool-of-Addresses

Fig. 2: Sequence diagrams for the three strategies.

1. QUIC server is triggered for migration. Depending on the approach used, either QUIC client or the migration management notifies QUIC server of an impending container migration.

2. Before container migration begins, QUIC server sends a SERVER MIGRATION frame to QUIC client, advertising that the server is going to migrate and notifying the client with the new address on which QUIC server is going to listen after migration.

3. QUIC client acknowledges the packet containing the SERVER MIGRATION frame to QUIC server, which is still listening on the old address.

4. Once the the server receives the acknowledgement to the SERVER MIGRATION frame, the container migration to the destination host can start. From this point on, the client does not know when the server will stop being reachable at the source host nor when it will be available at the destination one. We thus consider two alternative approaches: Proactive, the client assumes the server is no longer active on the source host and immediately considers the new address as the active one; and Reactive, wherein the client considers the old address to be

active until a packet-loss occurs. In both cases, as soon the new address is set as active, the client moves to the next phase.

5. QUIC client goes into a probing state in which it probes the new address of the server. Probing is a standard QUIC mechanism that is used to test liveness of the other endpoint. Probing can be performed by sending a packet that always contains a PING frame and possibly includes frames of the previously lost packet.

6. QUIC server replies to the probe from the new address on which it is now listening. The packet sent by QUIC server can be either probing or non-probing, depending on the packet sent by QUIC client in step 4. When QUIC client receives a non-probing packet from QUIC server, it sets the new path to the server as primary and exits the probing state.

7. If the new path to the server has not been validated before, QUIC client starts path validation, in accordance with QUIC specification. Therefore, it sends a PATH CHALLENGE frame to QUIC server, possibly along with other non-probing frames.

8. QUIC server replies with a packet that includes a PATH RESPONSE frame and possibly other non-probing frames.

9. QUIC client checks whether path validation is successful or not. If it is successful (as in the case reported in Fig. 2a), 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.

The rationale behind the two alternatives listed at step 4 is to offer a flexible support to the container-migration technique in use. As we explained in Section 2, techniques such as cold or post-copy immediately stop the container execution at the source node, whereas pre-copy and hybrid techniques leave it running while the state transfer is in progress. In the following we will refer to the above alternatives as Proactive Explicit (P-Explicit) and Reactive Explicit (R-Explicit), and we will compare their performance in Section 4, considering four container migration techniques.

3.3. Pool-of-Addresses

This strategy is designed to cope with situations where the set of possible addresses to which the server can migrate is well-known in advance. For instance, this may be possible in some small-scale orchestration scenarios or in mobility scenarios in which the area where the user's device moves is known. This pool of destination addresses can be either configured in a container base image or provided as a parameter when a container is launched. In the first case, all containers spawned from that image will share the same pool of addresses; in the second case, instead, each container may be provided with a different pool of addresses. The PoA strategy takes inspiration from the preferred-address mechanism (see Section 2.2) to let QUIC server notify QUIC client, during connection establishment, with the pool of possible migration addresses. Unlike the preferred-address case, however, QUIC server can migrate to any of those addresses in any moment, without previously informing QUIC client. More specifically, during connection establishment, QUIC server sends to QUIC client a POOL MIGRATION ADDRESS frame, which is defined in our design, for each address from the pool. QUIC client saves these addresses in a local pool. In any moment, the container can be migrated to another server machine without QUIC server or QUIC client being previously informed of that. When QUIC client is not able anymore to reach QUIC server on its current primary address, it probes addresses from the pool to find where QUIC server has migrated. QUIC client includes also the current primary address of the server among the addresses to be probed. This is done because QUIC server may not respond on the primary address due to network congestion rather than due to container migration. Moreover, it may happen that the QUIC client probes the correct server address *before* the container is available on the destination host. This could occur depending on various factors, including the container size, the connection speed between edge nodes, or the container migration technique in use. To prevent the client considering the server unreachable, the QUIC client probes the addresses in the pool cyclically until it finds the right one. It is worth noting that the QUIC client could be configured to probe addresses from the pool following a given criterion, e.g., a priority-based selection. This would allow one to limit the number of failures and thus to speed up the selection of the right address. Fig. 2b presents the sequence diagram of a successful connection migration realised through the PoA strategy. This strategy consists of the following steps:

1. During the initial handshake, QUIC server provides QUIC client with a set of addresses to which the server can possibly migrate during the connection. For each address inside the pool, QUIC server sends a POOL MIGRATION ADDRESS frame to QUIC client. The latter acknowledges these frames. Once connection between client and server is established, it works without any further changes until the event of migration.

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

3. After a packet to the server gets lost, QUIC client considers the possibility that QUIC server has migrated. Therefore, it probes the addresses from the pool. At the same time, it also probes the current primary address of the server to cope with the possibility that QUIC server does not respond due to network congestion rather than container migration. Probing in this strategy works in the same way as described in step 4 of the Explicit strategies.

4. QUIC server, which is now listening on the new address, replies to the probe sent by QUIC client. From now onward, the PoA strategy works in the same way as the Explicit ones. The packet sent by QUIC server can be either probing or non-probing, based on the packet sent by QUIC client in step 3. When QUIC client receives a non-probing packet from QUIC server, it sets the new path to the server as primary.

5. If QUIC client has never validated the new server address before, it must perform path validation. Hence, it sends to QUIC server a PATH CHALLENGE frame, which can be encapsulated in the same packet with other non-probing frames.

6. QUIC server replies with a PATH RESPONSE frame, possibly sent in the same packet with other non-probing frames.

7. Fig. 2b shows a situation in which path validation is successful. In this case, QUIC connection is migrated to the new address of the server. Otherwise, as specified in the QUIC draft, 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.

3.4. Implementation

We implemented the three proposed strategies by extending `aioquic`, an open-source QUIC implementation written in Python. Our implementation is publicly available on GitHub[23]. Our design does not break QUIC specifications, nor our extensions undermine `aioquic` implementation. To verify this, we validated our implementation by performing interoperability tests with other QUIC implementations. For space reasons, we do not include implementation nor validation details in this work. However, these details can be found in the preliminary version of this work [18].

4. Performance Evaluation

In this section, we evaluate our QUIC-based solution in its different flavours. To this purpose, we first describe the testbed that we used and then report the experiments that we performed. Besides, we compare our solution against one that uses TCP at transport layer and closes the TCP connection at the end of each request – CRIU indeed requires all TCP connections to be closed before migration to a different subnet starts. As soon as container migration terminates, the DNS record for the service is updated. When the DNS record in the local cache of the client expires, the latter can get the new IP address of the service, and establish a new TCP connection. We set the Time To Live (TTL) of the DNS record in the client local cache to 60 s, as this is a common value for dynamic host records [21]. We used BIND as DNS server.

We implemented two synthetic HTTP applications – one over QUIC and the other over TCP – which gave us freedom to tweak their parameters (e.g., the size of container state, the pattern of client requests) thus to cover different scenarios. Both the applications consist in a client issuing POST requests to a server. Each request contains a 1 kB payload, and the response from the server echoes the payload back to the client. In our evaluation, we consider the following metrics:

- *Service time* is the time between the beginning of a client request and the reception of the corresponding response from the server.
- *Container migration time* is the time to complete a container migration. It is the sum of the duration time of all the phases involved in the migration process, as reported in Section 2.1.

[2] `https://github.com/kruviser/aioquic-explicit_UniPisa`.
[3] `https://github.com/kruviser/aioquic-PoA_UniPisa`.

- *Container downtime* is the time during which the container is not up and running. It is sum of the Dump, Dump transfer, and Restore phases duration times.
- *Container migration overhead* is the amount of data transmitted during container migration.

The following performance evaluation aims at assessing how our QUIC-based connection-migration solution impacts the application. We quantify this impact by measuring the service time experienced by the client when connection migration is performed. As better explained in the next sections, we consider two highly different request patterns for the purpose, namely *sporadic requests* and *continuous back-to-back requests*. We are aware that many other request patterns do exist; however, in our analysis we are interested at evaluating our solution under two extreme cases: (i) one in which only connection migration impacts service time, which we reproduce using sporadic requests; (ii) the other in which both connection and container migration impact service time, which is obtained using continuous back-to-back requests.

4.1. Description of the Testing Environment

We ran our experiments on a realistic testbed, resembling the configuration of a small-scale edge scenario. As we show in Fig. 3, the testbed includes one end device, as the client, and two edge nodes, as the source and destination nodes. The latter are Qotom mini-PCs with a Quad-core Intel Celeron CPU at 1.99GHz, 8GB of RAM, and 58GB of disk, running Ubuntu 18.04 with Linux kernel 5.6.7. The end device is an ASUS Zenbook UX331UN laptop with an Intel i7-8550U CPU at 1.8GHz, 16GB of RAM, and 20GB of disk, running Ubuntu 20.04 and Linux kernel 5.4.0. As far as connectivity is concerned, the two edge nodes are connected using Gigabit Ethernet through a Netgear GS108E switch; they also act as Wi-Fi access points for the end device. The server application runs as an

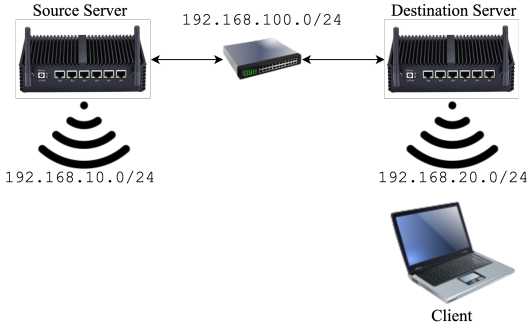


Fig. 3: Testbed configuration.

Alpine 3.7 runC container first on the source edge node and then migrates to the destination one. The client runs the client application and connects through Wi-Fi to the destination edge node. We use runC 1.0.1 as container runtime, CRIU 3.14 for the checkpoint/restore functionality, and `rsync` 3.1.2 to transfer the container state from source to destination. We performed ten repetitions for each experiment, and all the following results include a 95% confidence interval, when visible.

4.2. Scenario 1: Sporadic Requests

In this scenario, we consider a client issuing sporadic requests to a server, with a period of 10 s. Container migration starts after the second request is completed and terminates before the third request begins. This scenario allows us to measure service time *after* container migration has terminated, hence with an impact only from the QUIC connection migration procedure. In other words, we can evaluate the performance of our connection migration strategies with no influence from the migration of the container state. To this purpose, we migrate the container using only the cold technique. Container migration transfers 28 MB overall.

Fig. 4 presents service time impacted only by connection migration, using the different strategies. Besides, it reports two baseline cases – QUIC baseline and TCP baseline – wherein no container nor connection migration is performed. The P-Explicit configuration performs the best among all the migration options and has similar performance to nominal QUIC. TCP nominal pays the setup of a new connection at each request through the three-way handshake, resulting in at least 2 RTTs worth of delays. We recall that the container migration procedure cannot start if the container itself has any open TCP connection. The R-Explicit strategy, instead, performs worse than the P-Explicit, as a packet has to be lost before updating to the new address of the server. With regard to the PoA strategy, we considered an increasing values of the pool size (i.e., from one to three addresses). The QUIC client randomly selects addresses from the pool for probing and may also

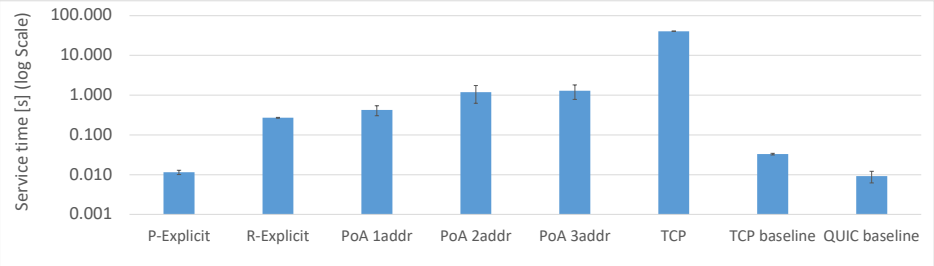


Fig. 4: Service time after container migration has terminated.

probe the old address of the server (see Section 3). This means that a pool of M addresses may lead to a maximum of M+1 addresses being probed. As we can see, the PoA strategy performs worse than the Explicit strategies, on average. This is also true in case of a one-address pool, which sometimes leads to two addresses being probed. Furthermore, the greater the pool size is, the worse is the performance of the PoA strategy because more addresses are probed on average. We note that the confidence intervals in Fig. 4 picture the variability in the number of attempts to find the correct server address within the pool. To conclude, performance of TCP after container migration is significantly worse than with any of the strategies based on QUIC. The TCP-based approach takes indeed 40 s on average to establish a new connection to the migrated container. This is given by the fact that the request impacted by the migration is issued on average after 20 seconds from the preceding DNS query and the TTL of the DNS record is 60 seconds. While it might be possible to choose a better value to optimize the performance in this case, we notice that in general this should be done dynamically to adapt to each specific scenarios.

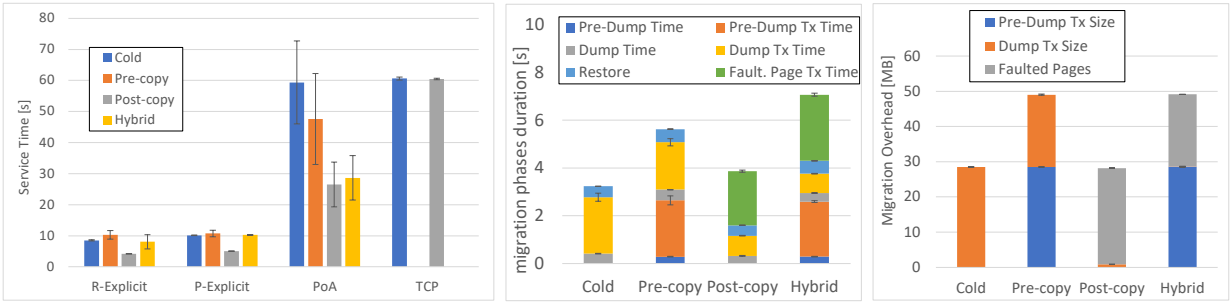
4.3. Scenario 2: Continuous back-to-back requests

We now consider a scenario wherein the client issues a request as soon as it receives the response to the previous one. This way, we aim at evaluating the service time when this is directly affected by both container and connection migration. Given that in this scenario container migration plays an important role, we investigate it under the four migration techniques described in Section 2.1 and considering two different sizes of containers in terms of memory footprint – a small size of 28 MB and a large one of 280 MB, which will have a direct impact on the container migration procedure. We highlight that a memory footprint of 280 MB allows us to test our solution under challenging conditions, considering that memory-intensive applications present an in-memory state that is around 100 MB [19]. With this second test scenario, we can analyse the interplay between container and connection migration, both studied under different flavours. For the PoA strategy, we consider only a pool size of three addresses. As in the previous scenario, container migration is triggered at the second request performed by the client. However, in this case, it can also have an impact on a request that is later than the third one: as we will see in the following discussion, this actually depends on the combination of container and connection migration strategies being used. As far as the TCP-based solution is concerned, we consider only cold and post-copy techniques, as

all TCP connections must be closed when CRIU checkpoints the container state. We can do this in cold and post-copy techniques by letting the client wait a short amount of time before issuing the third request, thus letting CRIU successfully checkpoint the container. Pre-copy and hybrid techniques, instead, include a pre-dump phase that makes it hard to predict the onset of container downtime, and thus to make the decision on when to avoid opening a new TCP connection.

We firstly analyse results obtained with the small-size container. Fig. 5a shows service time results. Fig. 5b depicts the duration of the phases of container migration. Fig. 5c, instead, reports the container migration overhead. By looking at Fig. 5a, it is easy to note how service times are considerably higher than the baseline cases as well as than results of the first scenario. This means that container migration has a non-negligible impact on service time. Going into details, R-Explicit and P-Explicit perform similarly, with the former being slightly better. This can be ascribed to the fact that QUIC congestion control doubles the delay between retransmissions after a packet is lost. P-Explicit loses more packets than R-Explicit in this scenario due to two main reasons. Firstly, container migration takes few tens of milliseconds to start after it is triggered. During this period of time, R-Explicit can still issue requests to the container, whereas P-Explicit immediately starts losing packets because it tries to reach the container at destination while this is still running at the source. Secondly, P-Explicit does not let the client contact the container during the pre-dump phase of pre-copy and hybrid techniques – the client perceives a downtime throughout that phase. R-Explicit, on the other hand, can still contact the container on the source server during that phase or at least during part of it. With the R-Explicit strategy, the client can in fact start perceiving downtime at a random point in time in the middle of the pre-dump phase. This occurs when QUIC client loses a packet (which is more likely while the network is being used for container migration) and does not receive anymore packets from the container on the old address – e.g., because the lost packet is the first of a new client request. All the above considerations will be more evident with the large-size container, as the pre-dump phase is longer in that case.

For what concerns the PoA strategy, we make the following considerations. Similarly to R-Explicit, PoA lets the client contact the container during the pre-dump phase or part of it. In particular, PoA performs better than R-Explicit from this point of view, as it can more easily revert to the old address of the container and continue communicating with it when a packet is lost during the pre-dump phase. In fact, it reverts to the old address when it either receives a packet from that



(a) Service time **(b)** Container migration time **(c)** Container migration overhead

Fig. 5: Scenario 1 results, for a small-size container

address (as the R-Explicit does) or when it probes the old address, during the probing phase of addresses in the pool, and receives a response. However, as shown in Fig. 5a, the PoA still performs worse than the Explicit strategies. The reason is that the pre-dump phase is too short with a small-size container, as shown in Fig. 5b, to make that positive characteristic of the PoA strategy emerge. The fact that PoA might need to perform several attempts to find out the new address has a greater impact on its overall performance, with a small-size container. This is much more evident under the cold migration technique, which does not have a pre-dump phase and has a much longer downtime than the post-copy technique. Finally, the width of confidence intervals is greater in case of: (i) pre-copy and hybrid techniques for R-Explicit and PoA because of the uncertainty in the beginning of the client-perceived downtime during the pre-dump phase; (ii) the PoA strategy, due to the variability in the number of attempts that are necessary to find the container at the destination server.

Analysing Fig. 5c, it is possible to note that pre-copy and hybrid techniques transmit nearly 50 MB on average against the 28 MB of cold and post-copy techniques. This translates in longer container migration times for the former two techniques, as shown in Fig. 5b. Besides, this additional time corresponds to the downtime for pre-copy migration.

To conclude with the analysis of the small-size container case, we focus on the results of the TCP-based solution. This solution lets the client establish a new connection to the migrated container after 60 s on average, regardless of the container migration technique in use. We highlight that the TTL of the DNS record could be optimised to improve the performance of the TCP-based solution, even though with the limitations outlined at the end of the first scenario. Overall, the TCP-based approach performs worse than both the R-Explicit and P-Explicit. It still performs worse than PoA

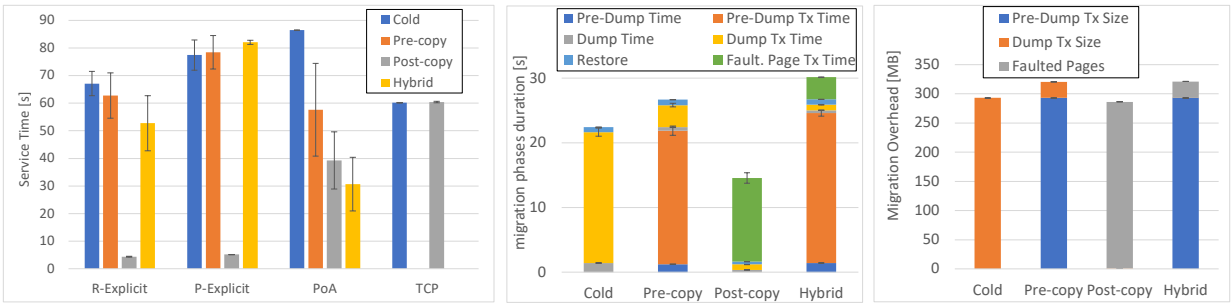
when using the post-copy technique, and better than that when cold migration is used.

We now analyse the results for the large-size container. Fig. 6a presents service time results. Fig. 6b reports the duration of phases of container migration; while Fig. 6c depicts the overhead of container migration. By looking at Fig. 6a, it is possible to note how the same considerations made for a small-size container are more evident.

As discussed before, R-Explicit and P-Explicit perform similarly in the small-sized container case. However, in case of a large-size container, the higher volume of data transmitted during migration and the higher migration times make the difference between the two strategies more evident, with R-Explicit performing better than P-Explicit (see Fig. 6a). Only under post-copy migration technique, the two strategies perform similarly, because post-copy technique has a short downtime and does not have a pre-dump phase.

When a large-size container is migrated, the PoA strategy performs better than R-Explicit in handling packet loss and reverting to the old address of the container while this is still in the pre-dump phase. In case of a small-size container the effects of this behaviour were not so evident. However, when a large-size container is migrated, i.e., the pre-dump phase lasts longer, this characteristic of the PoA strategy stands out, partially compensating the performance degradation in PoA due to the search of the new address among those in the pool. Overall, the result is that PoA with pre-copy technique performs similarly to R-Explicit and overtakes P-Explicit, in case of pre-copy. With hybrid technique, instead, PoA performs better than both the Explicit strategies. Yet, performance with cold and post-copy techniques are still lower than the equivalent with the Explicit strategies, since those migration techniques do not have a pre-dump phase.

To conclude, when cold migration is considered, the TCP-based approach exhibits better per-



(a) Service time **(b)** Container migration time **(c)** Container migration overhead

Fig. 6: Scenario 2 results, for a large-size container

formance than the other strategies. This is because the QUIC-based strategies suffer from the high downtime of cold migration, which causes a considerable increment of the client-side retransmission delay. On the other hand, the TCP-based solution is able to connect to the migrated container as soon as the client updates the DNS record in its cache (i.e., after 60 s). However, we highlight that all the QUIC-based strategies show non-negligible performance improvements with respect to the TCP-based approach in case of post-copy migration.

5. Related Work

Service migration between hosts has proved over the years as a powerful way to achieve fault tolerance and load balancing. More recently, service migration has been considered as a valid approach to support mobility of end devices in edge computing environments: when a user moves away from their edge service, the latter is migrated to preserve proximity to the user. In this context, several works propose methods to efficiently transfer the state of a service (typically encapsulated in a container) from one edge server to another [19, 22, 23] or evaluate the performance of migration of service state at the network edge [7]. In this work we tackled the related problem raised by service migration at the network edge. When a service migrates across different networks, it undergoes a change of IP address, which causes problems of service reachability after migration and termination of active TCP connections (if any). Several works from literature propose solutions at different layers of the protocol stack to seamlessly hold connectivity or migrate ongoing connections [24].

At the network layer, most of the solutions preserve connectivity by letting the service at the new location be reachable through the old IP address. This can be done through network virtualisation techniques, using ad-hoc protocols, or leveraging forwarding paradigms alternative to IP. Some works extend LAN across data centres to let a VM or container keep its network configuration (e.g., IP address) at the destination server [9, 10]. Mobile IP and LISP are layer-3 solutions that allow an IP address to move across networks. Some works leverage these protocols to achieve seamless connectivity holding towards a migrated VM [11, 25, 26]. Authors in [27] present an application-layer architecture that creates isolated virtual networks with their own IP address space. Finally, some authors propose approaches based on Software Defined Networking (SDN) for connectivity holding. SDN gives birth to highly programmable and flexible networks where network traffic can be redirected after a service has been migrated to a new host [28, 29].

Other solutions intervene at the application layer. Authors in [12] discuss a complex software platform to let containers be reachable after migration between cloud data centres. Specifically, this solution is designed only for web-based applications and consists of an HTTP proxy that holds incoming connections from clients while the container is being migrated. The proxy continuously polls the destination host and sends an HTTP redirect message to clients once the container has resumed execution at destination; meanwhile, the DNS entry of the service is updated with the new IP address. An interesting solution comes from MQTT. This protocol identifies connections between a client and the MQTT broker by means of a client identifier that is independent from the IP addresses of the endpoints. Therefore, MQTT brokers and clients can move across the network while preserving their active connections [30]. However, this approach is specific to MQTT-based applications. Also ETSI Multi-access Edge Computing (MEC) specifications define an API for application relocation to support mobility of users at the edge [13]. However, ETSI MEC does not define mechanisms to transfer the service state nor solutions to preserve active connections.

All the above works present some limitations in terms of overhead and complexity, which can be mainly ascribed to the need for auxiliary infrastructures or ad-hoc protocols at different layers of the protocol stack. This makes those solutions complex to deploy and may introduce non-negligible performance issues due to, e.g., triangular routing or propagation delays of DNS entries, which may be not tolerable especially in edge computing scenarios.

More related to the solution proposed in this work, some previous works focus on solutions at the transport layer and, more specifically, based on TCP. Authors in [31] stop a container with an active TCP connection and, in a later moment, resume both the container execution and its ongoing TCP connections on the same server machine. Moreover, they test container migration to a destination host. However, in that case, the container does not have open TCP connections. We also highlight that CRIU provides a functionality known as *–tcp-established* [32] that allows to migrate active TCP connections of a container. Nonetheless, one has to make sure that the container maintains the same IP address. In [33] and [34], authors develop an end-to-end mechanism that allows endpoints of an active TCP connection to seamlessly negotiate a change in IP addresses. However, it is worth noting that this mechanism was proposed in the early 2000s and has never been included as part of TCP implementation; therefore, applications cannot enjoy it.

To the best of our knowledge, Multi-Path TCP (MPTCP) is the only existing solution from the

literature that has been leveraged at the transport layer to migrate an active connection. MPTCP [35] was designed as an extension of TCP to unleash the full potential of *multi-homed* devices. Such devices are equipped with several network interfaces (e.g., cellular and Wi-Fi). MPTCP aims at letting these devices simultaneously use their multiple interfaces within a single connection, thus to aggregate bandwidth, balance load, and improve overall performance. Therefore, a MPTCP connection can result in one or more subflows, where each subflow is a regular TCP connection identified by a 4-tuple *<IP address, port>*. In MPTCP, this capability has been exploited, as side benefit, to migrate a MPTCP connection from one interface to another. Connection migration is indeed not possible for TCP, but MPTCP solves this problem even though it requires at least two network interfaces. Therefore, some works [36, 37] employ MPTCP to migrate connections of a mobile device that undergoes wireless handover. Other works leverage MPTCP between the source and destination hosts to enhance performance of container migration at the edge [38, 39] – which is orthogonal to our solution. Finally, authors in [40, 41] migrate a VM across cloud data centres and leverage MPTCP to migrate active connections of the VM.

It is worth noting that a multi-path version of QUIC exists - under the name of MPQUIC - [42] which extends QUIC having the purpose of leveraging multiple network interfaces of multi-homed devices, in a similar manner to what MPTCP does with respect to TCP. MPQUIC aims at reusing the mechanisms already provided by QUIC, including path validation and connection migration [42]. As a result, MPQUIC can leverage QUIC mechanisms, including our extension, to migrate connections.

In Section 4, we experimentally compared our work against an approach that migrates a container and updates the DNS record for the migrated container to let its client establish a new TCP connection towards the container on the new IP address. We dedicate the next Section 5.1 to compare analytically our solution against the one based on MPTCP. To this purpose, we consider the work in [40] as it is more recent and detailed than [41].

As a final remark, we highlight that our solution based on QUIC presents two major advantages with respect to related works in the field. Firstly, it avoids the complexity and overhead due to the need of deploying additional functions in different parts of the network and at one or multiple layers of the protocol stack (e.g., network virtualisation, specialised forwarding, dedicated protocols, cooperation between source and destination servers, etc.). Instead, our proposed solution exclusively

relies on direct interaction between the two endpoints of the connection. Secondly, our reconnection mechanisms are restricted to the transport layer, hence ensuring transparency of migration to QUIC-based application logic implementing both the client and the service. Finally, to the best of our knowledge, no other previous work evaluates the interplay of connection migration mechanisms with the different container migration techniques, as we do in Section 4.

5.1. Comparative analysis with MPTCP

In this section, we perform an analytic comparison between our solution based on QUIC and a solution based on MPTCP, which is described in [40]. We do not experimentally compare these two approaches because implementing the MPTCP solution is not straightforward. The code implemented by authors in [40] is not publicly available, and, in their paper, they report a number of non-trivial implementation issues that they encountered. Moreover, there is currently no reference

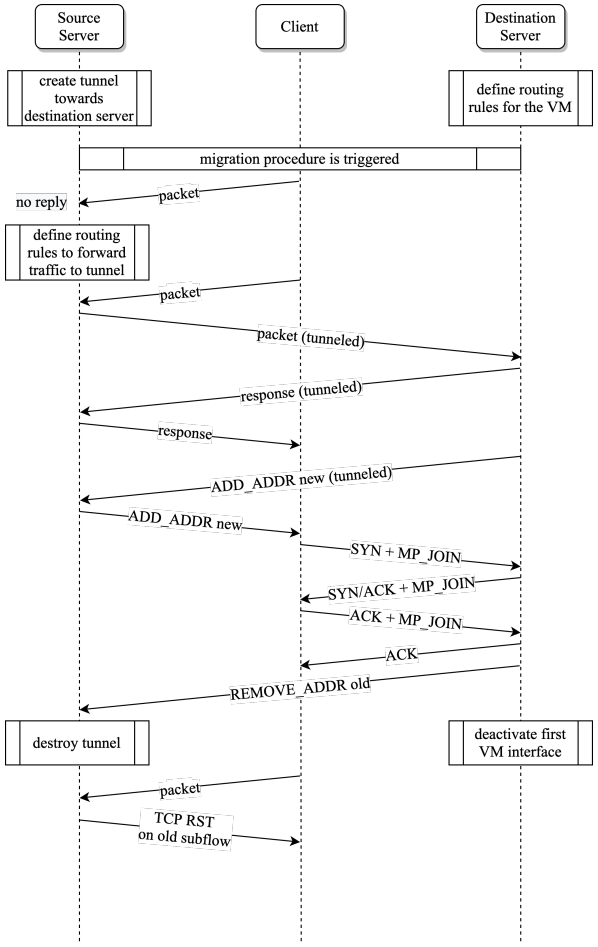


Fig. 7: Sequence diagram for the MPTCP strategy.

in literature on using MPTCP as transport layer for containers (either fixed or migrating), which makes the integration between these two technologies uncertain. As a result, we chose to carry out an analytic comparison. This consists in deriving a sequence diagram for the MPTCP approach, which is possible thanks to the design details reported in [40]. Then, we compare the QUIC and MPTCP solutions in terms of: (i) the RTTs that are necessary to let the client start communicating with the server on the new path; (ii) the RTTs that are necessary to complete the connection migration procedure. Fig. 7 depicts the sequence diagram for the MPTCP solution, which consists of the following steps:

1. Before VM migration starts, a GRE tunnel is created between the source and the destination hosts. Besides, routing rules are proactively configured on the destination host for both traffic incoming from and outgoing to the VM that is going to be migrated soon. The tunnel lets the VM, after migration, communicate with the client over the original subflow (passing through the source host). This is necessary, as MPTCP allows new IP addresses to be communicated through existing subflows only. However, it is worth noting that the tunnel introduces complexity and transforms this solution in a non-end-to-end one.

2. After VM migration starts, the client starts losing packets to the old address.

3. After VM migration completes, routing rules are defined at the source host to forward traffic to the tunnel. On the destination, a second interface of the migrated VM is activated. This is a second limitation of this solution. The VM is assumed to be multi-homed and have at least two interfaces, one for the subflow on the old address and one for that on the new address.

4. Communication between the client and the VM is restored through the old subflow. Traffic is triangulated, as it first has to reach the source host to be then forwarded through the tunnel to the destination one.

5. The VM sends a MPTCP *ADD_ADDR* option on the existing subflow to advertise the new IP address to the client.

6. The client starts a *MP_JOIN* procedure to create a new subflow on the new VM address. After three ways, the subflow is considered as *pre-established*. However, application data can

be sent on the new subflow only after reception of the final ACK. From now on, the new subflow is *established* and client and VM can communicate over it.

7. To complete the connection migration procedure, it is necessary to close the old subflow. To this purpose, the VM sends a MPTCP *REMOVE_ADDR* option on the old subflow. Hence, the tunnel is destroyed on the source host, and the first interface of the VM is deactivated. The next time the client sends a packet on the old subflow, it receives a TCP RST and considers the old subflow as definitely closed.

8. The MPTCP connection is now successfully migrated.

By analysing the sequence diagrams from Fig. 2a and Fig. 7, we can qualitatively compare the QUIC and MPTCP solutions. We indicate the client with C, the server on the source host with S, and the server on the destination host with D. For what concerns the start of communication between client and server on the new path, our solution takes $1RTT_{C-S}$. The MPTCP-based approach instead takes $2RTT_{C-D} + 1/2RTT_{C-S} + 1/2RTT_{S-D}$. With regard to completion of the whole connection migration procedure, our solution takes $1RTT_{C-S} + 2RTT_{C-D}$. The MPTCP approach instead takes $2RTT_{C-D} + 3/2RTT_{C-S} + 1RTT_{S-D}$. As a result, our solution outperforms the one based on MPTCP.

6. Conclusions

Microservices are a powerful architectural style to deliver software services to end users. Nowadays, they are mostly deployed in the form of containers, and are an attractive solution for edge-based deployments, wherein the service is brought closer to the end user. However, the deployment of edge-hosted microservices poses several challenges, as users at the network edge are often mobile, thus requesting the microservices to *migrate* accordingly across the edge nodes. In this work, we presented our solution to extend QUIC towards server-side connection migration to support stateful container-migration at the network edge. More specifically, we proposed three strategies (i.e., P-Explicit, R-Explicit and PoA) to achieve this purpose. We developed our solution on top of an existing QUIC implementation, namely `aioquic`, and we validated it through extensive testing. We carried out a performance evaluation on a realistic edge-based testbed, comparing our solution

against a TCP-based one. We analysed the behaviour of the considered alternatives for four different container-migration techniques, highlighting the interplay among the two mechanisms. Results have demonstrated that post-copy performs better than the other techniques both in terms of downtime and in terms of volumes of transferred data, especially when used in conjunction with the R-Explicit or P-Explicit strategies. Besides, P-Explicit performs better than R-Explicit when only connection migration impacts service time. However, when also container migration has an impact, R-Explicit outdoes P-Explicit, especially in case of containers with a big memory footprint. Results have also shown that, in general, PoA performs worse than the Explicit strategies. Nevertheless, it shows better performance when big-sized containers are migrated using the hybrid technique. Finally, our solution represents overall a better alternative to solutions based on TCP+DNS and MPTCP. As a future work, we plan to integrate our solution on a fully fledged edge platform, e.g., based on ETSI MEC, and to explore the interactions of our solution with a centralised orchestration mechanism.

Acknowledgements

This work was partially supported by the Italian Ministry of Education and Research (MIUR) in the framework of the CrossLab project (Departments of Excellence).

References

[1] P. Jamshidi, C. Pahl, N. C. MendonÃ§a, J. Lewis, S. Tilkov, Microservices: The journey so far and challenges ahead, IEEE Software 35 (3) (2018) 24–35.

[2] P. Sharma, L. Chaufournier, P. Shenoy, Y. C. Tay, Containers and virtual machines at scale: A comparative study, in: ACM Middleware, 2016.

[3] U. Ramachandran, H. Gupta, A. Hall, E. Saurez, Z. Xu, A case for elevating the edge to be a peer of the cloud, GetMobile: Mobile Comp. and Comm. 24 (3) (2021) 14–19.

[4] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, D. Sabella, On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration, IEEE Comm. Surveys & Tutorials 19 (3) (2017) 1657–1681.

[5] C. Puliafito, E. Mingozzi, C. Vallati, F. Longo, G. Merlino, Virtualization and migration at the network edge: An overview, in: IEEE SMARTCOMP, 2018, pp. 368–374.

[6] S. Wang, J. Xu, N. Zhang, Y. Liu, A survey on service migration in mobile edge computing, IEEE Access 6 (2018) 23511–23528.

[7] C. Puliafito, C. Vallati, E. Mingozzi, G. Merlino, F. Longo, A. Puliafito, Container migration in the fog: A performance evaluation, Sensors 19 (7) (2019).

[8] A. Elgazar, K. Harras, Teddybear: Enabling efficient seamless container migration in user-owned edge platforms, in: IEEE CloudCom, 2019, pp. 70–77.

[9] T. Wood, K. K. Ramakrishnan, P. Shenoy, J. Van der Merwe, J. Hwang, G. Liu, L. Chaufournier, CloudNet: Dynamic pooling of cloud resources by live WAN migration of virtual machines, IEEE/ACM Trans. on Networking 23 (5) (2015) 1568–1583.

[10] A. J. Mashtizadeh, M. Cai, G. Tarasuk-Levin, R. Koller, T. Garfinkel, S. Setty, XvMotion: Unified virtual machine migration over long distance, in: USENIX ATC, 2014, pp. 97–108.

[11] E. Silvera, G. Sharaby, D. Lorenz, I. Shapira, IP mobility to support live migration of virtual machines across subnets, in: SYSTOR, 2009.

[12] T. Benjaponpitak, M. Karakate, K. Sripanidkulchai, Enabling live migration of containerized applications across clouds, in: IEEE INFOCOM, 2020, pp. 2529–2538.

[13] ETSI, Multi-access edge computing (MEC); application mobility service API, Tech. rep. (Jan. 2020).

[14] J. Iyengar, M. Thomson, QUIC: A UDP-based multiplexed and secure transport, RFC 9000, RFC Editor (May 2021).

[15] M. Bishop, Hypertext transfer protocol version 3 (http/3), Internet-Draft draft-ietf-quic-http-34, IETF Secretariat (Feb. 2021).

[16] D. Madariaga, L. Torrealba, J. Madariaga, J. Bermúdez, J. Bustos-Jiménez, Analyzing the adoption of quic from a mobile development perspective, in: ACM EPIQ, 2020, p. 35–41.

[17] L. Tan, X. Gao, W. Su, N. Li, W. Zhang, Connection migration in quic, Internet-Draft draft-tan-quic-connection-migration-00, IETF Secretariat (Oct. 2020).

[18] L. Conforti, A. Virdis, C. Puliafito, E. Mingozzi, Extending the quic protocol to support live container migration at the edge, in: IEEE WoWMoM, 2021, pp. 61–70.

[19] A. Machen, S. Wang, K. K. Leung, B. J. Ko, T. Salonidis, Live service migration in mobile edge clouds, IEEE Wireless Communications 25 (1) (2018) 140–147.

[20] CRIU, Live migration for containers is around the corner.
URL archive.fosdem.org/2015/schedule/event/livemigration/

[21] Dyn, FAQs - Time to Live (TTL), URL: help.dyn.com/ttl (Jan. 2021).

[22] L. Ma, S. Yi, N. Carter, Q. Li, Efficient live migration of edge services leveraging container layered storage, IEEE Trans. on Mobile Computing 18 (9) (2019) 2020–2033.

[23] K. Govindaraj, A. Artemenko, Container live migration for latency critical industrial applications on edge computing, in: IEEE ETFA, Vol. 1, 2018, pp. 83–90.

[24] F. Zhang, G. Liu, X. Fu, R. Yahyapour, A survey on virtual machine migration: Challenges, techniques, and open issues, IEEE Comm. Surveys & Tutorials 20 (2) (2018) 1206–1243.

[25] E. Harney, S. Goasguen, J. Martin, M. Murphy, M. Westall, The efficacy of live virtual machine migrations over the Internet, in: IEEE VTDC, 2007, pp. 1–7.

[26] S. Kassahun, A. Demessie, D. Ilie, A PMIPv6 approach to maintain network connectivity during VM live migration over the internet, in: IEEE CloudNet, 2014, pp. 64–69.

[27] X. Jiang, D. Xu, VIOLIN: Virtual internetworking on overlay infrastructure, in: Parallel and Distributed Processing and Applications, 2005, pp. 937–946.

[28] K. A. Noghani, A. Kassler, P. S. Gopannan, EVPN/SDN assisted live VM migration between geo-distributed data centers, in: IEEE NetSoft, 2018, pp. 105–113.

[29] J. Liu, Y. Li, D. Jin, SDN-based live VM migration across datacenters, in: ACM SIGCOMM, 2014, p. 583–584.

[30] C. Puliafito, A. Virdis, E. Mingozzi, The impact of container migration on fog services as perceived by mobile things, in: IEEE SMARTCOMP, 2020, pp. 9–16.

[31] P. Karhula, J. Janak, H. Schulzrinne, Checkpointing and migration of iot edge functions, in: IEEE EdgeSys, 2019, p. 60–65.

[32] CRIU, TCP connection, URL: criu.org/TCP_connection (Jan. 2018).

[33] A. C. Snoeren, D. G. Andersen, H. Balakrishnan, Fine-grained failover using connection migration, in: USENIX ITS, 2001.

[34] A. C. Snoeren, H. Balakrishnan, An end-to-end approach to host mobility, in: ACM Mobicom, 2000, p. 155–166.

[35] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, Tcp extensions for multipath operation with multiple addresses, RFC 6824, RFC Editor (Jan. 2013).

[36] O. Bonaventure, C. Paasch, G. Detal, Use cases and operational experience with multipath tcp, RFC 8041, RFC Editor (Jan. 2017).

[37] C. Paasch, G. Detal, F. Duchene, C. Raiciu, O. Bonaventure, Exploring mobile/wifi handover with Multipath TCP, in: ACM SIGCOMM Workshop on Cellular Networks, 2012, p. 31–36.

[38] Y. Qiu, C.-H. Lung, S. Ajila, P. Srivastava, Experimental evaluation of LXC container migration for cloudlets using multipath TCP, Computer Networks 164 (2019) 106900.

[39] Y. Qiu, C.-H. Lung, S. Ajila, P. Srivastava, LXC container migration in cloudlets under Multipath TCP, in: IEEE COMPSAC, Vol. 2, 2017, pp. 31–36.

[40] F. Le, E. M. Nahum, Experiences implementing live VM migration over the WAN with Multi-Path TCP, in: IEEE INFOCOM, 2019, pp. 1090–1098.

[41] C. Nicutar, C. Paasch, M. Bagnulo, C. Raiciu, Evolving the Internet with connection acrobatics, in: ACM HotMiddlebox, 2013, p. 7–12.

[42] Y. Liu, Y. Ma, Q. D. Coninck, O. Bonaventure, C. Huitema, M. Kuhlewind, Multipath extension for QUIC, Tech. rep., IETF (Oct. 2021).