An attempt at introducing Multipath in QUIC

Alessandro Celestino and Simon Pietro Romano
Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione
Università degli Studi di Napoli Federico II, Napoli, Italy
Email: alessandro.clestino@gmail.com, spromano@unina.it

***Abstract*—In this paper we present a proposal for the introduction of multipath functionality in the QUIC protocol. We will discuss how we designed such functionality by taking inspiration from state-of-the-art solutions like the one made available with multipath TCP. We will highlight how we leveraged QUIC unique features in order to overcome some of the well-known limitations of multipath TCP. We will also discuss how we implemented a basic version of multipath QUIC by modifying the existing code base made available by Google. We will present preliminary evaluation results allowing us to draw some interesting conclusions about our work, as well as to identify space for further improvements. We will finally point out the main lessons we learnt, by also making some reflections about ongoing standardization efforts within the Internet Engineering Task Force, with special reference to the QUIC protocol and its major milestones.**

***Keywords*—Multipath, Transport Protocols, QUIC, HTTP/2.**

I. Introduction

The ever-increasing utilization of middleboxes like NATs (Network Address Translators) and firewalls in the Internet has made both the development of new transport protocols and the extension of existing ones harder and harder. Such a fossilization at the transport layer has fostered interest around the use of UDP as an enabler for the development of new user-layer protocols which actually provide transport-level functionality. UDP can in fact be leveraged to somehow encapsulate such new protocols with the twofold objective of facilitating their deployment and allowing them to pass through NATs and firewalls.

QUIC (Quick UDP Internet Connections) is a perfect example of such new generation protocols and is currently being standardized by the IETF (Internet Engineering Task Force). QUIC is a transport-layer protocol that gets encapsulated inside UDP and which encrypts most of its headers and the entire payload as well. Thanks to that, it is capable to offer built-in privacy and security to the applications, by also keeping information that is visible to the endpoint separate from what is visible to the on-path devices. This approach naturally encourages a quick adoption of the protocol. QUIC aims at improving some crucial aspects of legacy protocols like TCP. It reduces connection establishment latency, as well as tackles the infamous head-of-line blocking issue by natively supporting stream multiplexing. It also looks after end-to-end security through the adoption of TLS (Transport Layer Security) version 1.3. The protocol has also been designed so as to support seamless mobility of network nodes, as well as improved loss detection and loss recovery. QUIC provides native support to HTTP/2 as the most likely application-level candidate. Other protocols with similar requirements will be nonetheless supported in the future.

Among the desired set of functions to be inserted in the protocol, *multipath* (i.e., the capability of transferring a stream of data across two or more different paths) definitely plays a major role. Multipath transmission can indeed be fruitfully leveraged also in those cases in which the end-user device makes available multiple interfaces (e.g., both WiFi and 3G), which can be jointly used in order to optimize communication in terms of both effective network utilization and improved end-user's experience.

This paper describes an attempt at introducing support for multipath transmissions inside QUIC. In Sec. II we will delve into QUIC protocol internals. In Sec. III we will focus on multipath transmission, by highlighting both its benefits and potential drawbacks. We will then propose, in Sec. IV, a novel solution for the introduction of multipath in QUIC. We will first focus on the design phase, by describing the necessary modifications to the original protocol structure. Then, we will describe on how we implemented our proposal by modifying the Google Chromium code base. In Sec. Vwe will present some preliminary performance results. We will conclude the paper in Sec. VI with an analysis of lessons learnt, as well as with some discussion on our directions of future work.

II. An introduction to the QUIC protocol

QUIC is an experimental transport-layer protocol initially developed by Google and proposed to the IETF for standardization in 2015. A dedicated IETF Working Group has been chartered in 2006 and is currently working on standardizing both the protocol internals and the mapping of application-layer protocols like HTTP/2 onto it. QUIC relies on UDP as the underlying transport means, which guarantees compatibility with legacy clients and middleboxes. Furthermore, QUIC enforces complete header authentication, as well as almost complete header encryption. It is hence capable to offer a security level which is equivalent to TLS. The protocol also drastically reduces connection establishment and transport latency, and implements advanced congestion control mechanisms. In the following we will briefly describe the main features of the protocol, by referring to the ongoing draft specification documents [1], [2], [3], [4].

In order to establish a secure end-to-end connection, QUIC relies on a handshake for the exchanging of security parameters, combined with a further handshake associated with transport parameters. QUIC connections are supposed to use, most of the times (i.e., for all but the very first connection towards a specified server), a so-called 0-RTT (*Zero Round Trip Time*) handshake. This means that data can be sent immediately following the client handshake packet, by re-using the same cryptographic material exchanged upon setting up

the first connection, with no need to wait for a reply from the server. QUIC also provides a reserved stream (with ID 0) which is specifically used to negotiate connection options.

A novel feature of the protocol is *stream multiplexing*, that is the simultaneous transmission of multiple streams across a single connection. While doing this, the protocol takes care of avoiding undesirable effects like the so-called head-of-line blocking. It hence guarantees that the loss of a packet just impacts the stream to which that packet belongs. Data received across the same connection, but associated with different streams, keep on being seamlessly reassembled and delivered to the intended destination. This is totally different than what happens with legacy protocols like TCP, for which the loss of a packet delays the delivery of all subsequent packets until the retransmission phase has been successfully completed.

When designing the protocol, a lot of attention has been devoted to congestion control and loss recovery. Each QUIC packet is assigned a unique packet number. This holds true both for new packets and for retransmissions. With this approach, there's no need to explicitly differentiate between acknowledgements associated with original transmissions and acknowledgements related to retransmitted packets. The acknowledgement packets are also used to carry back crucial information about the delay measured between the time when the packet has arrived at the receiver and the time when the acknowledgment itself is sent. This information, combined with the monotonically increasing packet number, allows for a reliable computation of the RTT experienced in the network.

With respect to flow control (both at the stream and at the connection level) QUIC draws inspiration from HTTP/2. A receiver advertises a certain byte offset for each stream for which it is willing to receive data. Once data are received and delivered on a stream, the receiver sends a WINDOW_UPDATE frame which increments the offset value, hence enabling the transmitter to send further data across that stream. On top of that, QUIC also implements flow control at the connection level, with the aim of limiting the overall amount of buffer that a receiver is willing to allocate for all of the streams belonging to the connection in question. Connection-level flow control works similarly to stream-level flow control, but it takes into account the total bytes delivered across the connection, as well as the highest offset among those advertised for all of the streams. Coming to security requirements, all QUIC packets are authenticated and the entire payload is encrypted. The parts of the header which are not encrypted get anyhow authenticated at the receiver's side, so to avoid data tampering and data injection. Some of the initial handshake packets, and the so-called *Version Negotiation* packet as well, are not encrypted. Though, their content gets verified once the connection has been established. Perhaps the most interesting concept introduced by QUIC is the "connection". A connection is identified by a 64-bit long Connection ID field, which is randomly generated by the client. Such a unique identifier decouples the connection from end-system information like IP address and port number. Connections can hence easily survive changes in those parameters (changes that might be ascribed to either updates to the translation rules applied by a NAT, or to the client's migration from one network to the other, with subsequent modifications to the IP address used). QUIC allows for an automated check of a client's identity, which is deemed to be valid provided that the cryptographic material used to both encrypt and decrypt packets does not change. With the mentioned approach, the Connection ID can also be used to allow a connection to "migrate" towards a different server address. The only thing that matters is, once again, consistency in the cryptographic information used for exchanging data.

III. MULTIPATH COMMUNICATION

Smart devices equipped with multiple interfaces are nowadays widely deployed. Though, even if such interfaces might be contemporarily used to better exploit Internet connectivity, their potential does not get disclosed due to the fact that the TCP/IP stack supports just a single interface for network-enabled communication. The scenario is lately changing thanks to the introduction of so-called multipath-enabled protocols, which allow for the transmission of data across multiple paths by leveraging more than one of the available device interfaces.

In a nutshell, multipath is a transmission technique allowing for the transmission and reception of data across multiple alternative paths. A path is a sequence of links connecting a sender and a receiver, typically identified by a 4-tuple formed by the two pairs "(IP address, port number)" associated, respectively, with the source and the destination. Multipath transmission entails a number of advantages, in terms of: (i) 1emphReliability, which is improved thanks to the fact that failures (as well as performance degradation) along one of the paths can be compensated by leveraging alternative paths which can keep a connection active; (ii) *Bandwidth aggregation*, since the overall achievable throughput can in principle be multiplied by the number of available paths between source and destination nodes; (iii) *Security*, which benefits from the inherent distribution of potentially sensitive information across the multiple available paths (a fact that obviously contrasts malicious actions like eavesdropping and packet sniffing).

A. Multipath TCP

"Multipath TCP" (MPTCP) definitely represents one of the most successful protocols supporting multipath. The main goal of MPTCP is to extend TCP (which has been natively conceived as a single-path communication protocol) in such a way as to make it capable of supporting transmissions across multiple paths. MPTCP is hence capable of optimizing use of network resources while at the same time improving end-users' experience thanks to the increased achievable throughput, as well as improved reliability in the presence of failures. A Multipath TCP connection provides a bidirectional stream of bytes between two hosts in a way that is fully compliant with legacy TCP, i.e., with no need for modifications at the application layer. With MPTCP, hosts can leverage multiple paths (when available), each associated with different IP addresses, to exchange segments belonging to the same MPTCP connection, which is seen by the application as a standard TCP-based logical communication channel. For what concerns the network layer, each MPTCP sub-flow is treated as an independent TCP stream, whose segments by the way also carry a new option type. MPTCP takes on the responsibility of creating, tearing down and managing the mentioned sub-flows in order to exchange data. The number of sub-flows included in a single MPTCP "connection" is not fixed a priori, as it can be dynamically adjusted during a connection's lifetime. When

creating an MPTCP connection, the same messages as those involved in a plain TCP handshake are exchanged, with the sole difference that SYN, SYN/ACK and ACK segments also carry the multipath option named MP_CAPABLE. This option is of a variable length and serves a twofold goal: (i) check whether the remote host supports Multipath TCP; (ii) allow the two hosts to exchange information needed to properly authenticate the creation of additional sub-flows.

IV. MULTIPATH IN QUIC

Due to the numerous limitations identified for MPTCP, a novel approach to multipath has lately been proposed. Such an approach looks like a potentially disruptive one, since it makes use of QUIC, rather than TCP, as the base protocol. Such a choice is justified by a number of considerations. QUIC is indeed independent from both the underlying hardware and the operating system. Both such factors make its deployment much faster. It also natively supports multiple streams. The introduction of multipath in this case would allow both stream and path multiplexing. Coming to security, QUIC makes use of TLS, hence guaranteeing data security by design even in the presence of multiple paths. Finally, QUIC requires much less overhead for connection setup when compared to TCP. In the light of the above considerations, we have worked on introducing multipath capabilities in QUIC. Starting from the code base made available by Google, which implements QUIC-enabled client-server communications within the Chromium browser, we have first designed and then implemented the desired multipath functionality.

A. Design phase

Drawing inspiration from multipath TCP, we have defined the multipath functionality in QUIC in such a way as to solve some of the MPTCP issues illustrated above, while at the same time aligning to the QUIC working group philosophy, as briefly summarized below: (i) place new functionality in user's space, so to foster rapid deployment of the protocol; (ii) minimize overhead due to the creation of a multipath connection, as well as to the addition of new paths; (iii) leverage TLS to guarantee security on each and every path. Both the design and the implementation of our proposal refer to the version of QUIC that is described in [5]. The version in question is in fact the one that was in place when we first started our work. It is also a version in which multipath was looked at as a fundamental feature to be natively supported by the protocol. At the time of this writing, there seems to be consensus in the working group with respect to the fact that multipath, though important, will be developed in future versions of the protocol, in the form of an 'extension'. Our objective is to create multiple paths within a single QUIC connection (as identified by its Connection ID), so to allow both client and server to send, receive and process packets across all such paths. When designing our solution, we have obviously taken inspiration from the MPTCP protocol. We have both adapted MPTCP to the use of the brand new transport made available by QUIC and tried to leverage QUIC's functionality in order to overcome MPTCP's known limitations.

During the connection establishment phase, QUIC makes use of a handshake for the exchanging of both security and transport parameters. During such a phase, clients can rely on the MULTIPATH_ENABLED header field in order to indicate their willingness to exploit multipath communication. In this way, we are able to reduce the overhead due to the three-way handshake which is typical of Multipath TCP. The newly devised handshaking process is illustrated in Fig. 1.

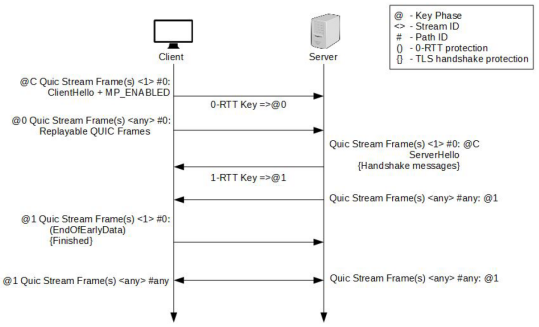

Fig. 1. QUIC handshake in the presence of Multipath

Once done with the transport and security parameters handshaking, both client and server can create multiple data transmission paths if multipath communication has been negotiated. In order to do that, they simply need to start sending data along a newly created path. Such data will contain the following information: (i) *PathID*, representing the chosen path identifier; (ii) *Multipath Flag*, always set to 1 for a multipath-enabled QUIC connection. Upon reception of the first (authenticated and encrypted) packet transferred along a newly created path, the receiving host will add the path number contained inside the 'PathID' field to the list of active paths associated with the connection in question. Hence, as soon as a new path is created, both endpoints will be capable of transferring data across it. If, on the other hand, we're just interested in advertising to the remote endpoint the availability of one or more additional paths, we can send an ad hoc designed *PING* frame across each of them. Such a frame, which will obviously have the multipath flag set, carries the chosen path identifier. After receiving such a packet, the remote endpoint will be allowed to send data along the available paths, as it happens with the ADD_ADDR functionality made available by Multipath TCP. If any of the two endpoints has either created or advertised an additional path, both of them can send and receive any frame belonging to any stream across any of the available paths. In this way it is possible to multiplex data communications both at the stream and at the path level. Two different levels of packet number are defined in order to allow for proper reordering of the packets: (i) a 'connection-level' packet number, keeping track of the sequence of packet numbers associated with all packets sent along all of the available paths; (ii) a 'path-level' packet number, indicating the sequence of packets sent along a specified path. Path-level packet numbers obviously must be associated with the corresponding connection-level packet numbers. This is achieved by leveraging the same mechanisms used for Multipath TCP.

B. Implementation phase

Given the above described design, the implementation phase has consisted in modifying the Google Chromium code base[1]. In this section we will first briefly describe the code

[1] https://www.chromium.org/quic

which allows transmission of data along a single path, by identifying data structures and entities that need to be either modified or replicated to enable multipath communications. We will then move to describing the modifications we made. We remark that our aim was to minimize impact on the existing code base. We hence decided to focus on the core functionality, by relying on a number of additional hypothesis. First, we assume that the client is the one who takes the responsibility of setting up multiple paths, by using a set of available IP addresses which get signaled to the server via PING frames. Second, we allow endpoints to send/receive packets along the created paths by either modifying the existing protocol entities or adding new ones when strictly needed. Third, we implement a simple round-robin technique when transmitting data along a set of available paths.

1) The Chromium code base: This software makes available a set of libraries allowing to implement client-server communication through the QUIC protocol. It has been developed to test the effectiveness of the new protocol, as well as to experiment with new features. As already anticipated, the release we refer to is the one mentioned in [5].

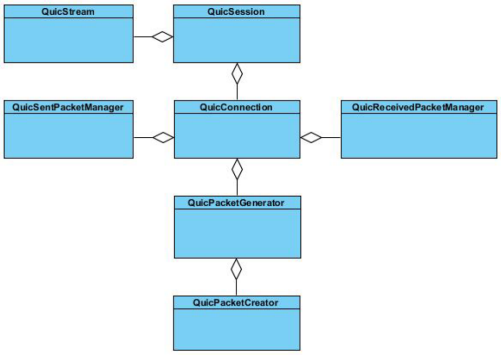

Fig. 2. QUIC communication classes

Fig. 2 shows a simplified class diagram containing the main classes involved in a QUIC communication session. The *QuicSession* object represents a QUIC session and looks after both stream and connection management. A QUIC session is also responsible for the mapping between QUIC and the related application-level protocol, as well as for connection flow control (via a dedicated class called *QUICFlowController*). *QuicStream* makes available methods and data structures that are needed in order to create, delete and manage QUIC streams. *QuicConnection* represents a single QUIC connection, as identified by its Connection ID. Such a class actually embeds the overall QUIC state machine associated with a QUIC connection and hence allows to properly manage data transmission, reception and processing. *QuicSentPacketManager* keeps track of all packets transmitted along a QUIC connection and implements a specific packet transmissions scheduling algorithm. Such a class also oversees packets acknowledgments and retransmissions, by storing data payloads and reusing them when needed. *QuicReceivedPacketManager* keeps track of QUIC connection state, based on received data packets. *QuicPacketGenerator* is responsible for generating packets on behalf of the QuicConnection: when a packet needs to be sent, the packet generator serializes it and hands it to the connection. Finally, *QuicPacketCreator* buffers data frames to be inserted inside the next packet scheduled for transmission, up to the point when either the maximum packet dimension is reached, or the QUIC transmission timer fires.

In order to enable multipath communications, we had to both replicate some of the above mentioned data structures (once per available path) and modify some classes. In the next subsection we will delve a bit into the details of such a process.

2) Code base modifications: The starting point here is the *QuicConnection* class, that is the main entity responsible for packets transmission, reception and manipulation. Since multipath transmission takes place at the connection level, no modifications were applied neither to the *QuicSession*, nor to the *QuicStream* class. We will briefly describe below how we updated the remaining classes, as well as how we added new classes which are needed to enable the newly introduced multipath transmission technique. Fig. 3 shows the updated class diagram.

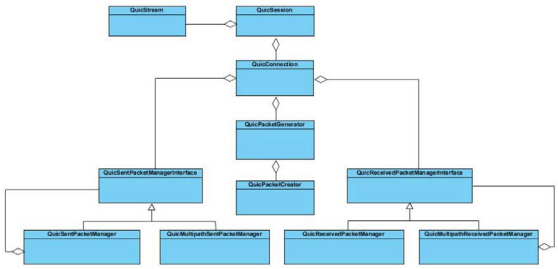

Fig. 3. QUIC multipath communication classes

As a preliminary operation, we had to create a brand new data structure called *QuicPathId*, an 8-bit integer keeping track of the path (among the set of available ones) used to transfer data. We have then identified those data structures which needed to be replicated for each and every available path in a multipath communication. The first such structure is the *QuicPacketWriter*, i.e., the class responsible for sending packets across the underlying UDP socket. We now instantiate as many writers as the number of active paths; such writers are stored in a dedicated array, whose index represents a specific PathId. We did the same also for the *peer_address* attribute containing the address of the remote peer.

Once multiple paths have been made available, packets transmitted along them will refer to different packet number spaces, which implies that acknowledgement packets will have to be managed independently on each and every path. We hence replicated, for every path, the following structures: (i) *ack_alarm*, a timer that fires every time a new ack needs to be sent; (ii) *ack_queued*, a boolean variable which is true when an ack needs to be sent back upon next transmission; (iii) *largest_seen_packet_with_ack*, representing the most recent packet sent by the peer and containing an ack.

In order to enable packet transmissions based on a round-robin policy, we added a new variable called *next_path* (whose type is QuickPathId) that gets incremented after each transmission and which is used to properly set, through the *SetCurrentPath* method of the *QuicPacketCreator* class, the path to be used.

To properly manage packets sent and received along a specified path, we had to redefine both the *QuicSentPacketManager*

and *QuicReceivedPacketManager* classes. The former looks after sent packets and includes both the transmission and the loss detection algorithm. In order to cope with the multipath case, we introduced a new class called *QuicMultipathSentPacketManager* that manages multiple *QuicSentPacketManager* instances (one per available path). We also introduced a higher level interface (*QuicSentPacketManagerInterface*) that is common to the two manager 'flavors' mentioned above. In this way, the former manager is instantiated if the *multipath_enabled* flag is set. In that case, a different QuicSentPacketManager will be created for each and every available path. The interface in question defines the same methods contained in the legacy QuicSentPacketManager class, but adds a new *QuicPathId* parameter to every method.

The same approach has been applied to the *QuicReceivedPacketManager*, by defining: (i) a new *QuicMultipathReceivedPacketManager* class taking on the responsibility to orchestrate the managers associated with packet receptions along the available paths; (ii) a new *QuicReceivedPacketManagerInterface* interface implementing exactly the same set of methods as the QuicPacketManager, with the addition of a QuicPathId parameter in input to every method. In this way, connection state is managed by the multipath manager and aggregates the set of states associated with the single paths, whose management is delegated to the corresponding QuicReceivedPacketManager instances.

Last but not least, the *QuicPacketCreator* oversees packet creation procedures. Among its tasks, we find the population of a packet's public header fields. In a multipath transmission, the creator must keep track of the current packet number associated with each and every path. This is achieved through a map that associates the last packet sent with the path along which it was actually transmitted.

V. Trials and Experimentations

In order to assess the correct behavior of our multipath QUIC implementation, we created the testbed depicted in Fig. 4. In such a testbed we have a client that sends requests to a remote server by also indicating the availability of multiple paths through the transmission of QUIC PING frames. Each path will be associated with a different UDP socket, whose source port parameter will be used to properly route packets along the set of available paths. The server will in turn be capable of managing packets coming from different paths. As to the other nodes, they will look after standard routing operations. The testbed has been realized as a fully virtualized network environment, by leveraging the Docker platform. To perform a comparative analysis between single-path and multi-path communication with QUIC, we have built two distinct Docker images hosting, respectively, the original Chromium code base and its multipath-enabled counterpart. Both containers are based on a Ubuntu 16.04 Linux distribution. To route packets inside the testbed we rely on four distinct router nodes. Also in this case, the chosen Linux distribution is Ubuntu 16.04.

In the aforementioned scenario we have run a set of experiments using QUIC. The main objective of such tests has been to measure the overhead introduced by the proposed multipath transmission technique. The presence of multiple paths, in fact, unavoidably makes data communication (transmission, reception and processing) more complicated. Furthermore, the currently implemented round-robin policy definitely represents the naivest scheduling technique one can put into place, since it does not allow for any smarter utilization of the set of available paths. We did not perform any experiment aimed at highlighting the obvious improvements associated with any multipath-enabled communication technique, like, e.g., fault tolerance or increased robustness and security of the communications.

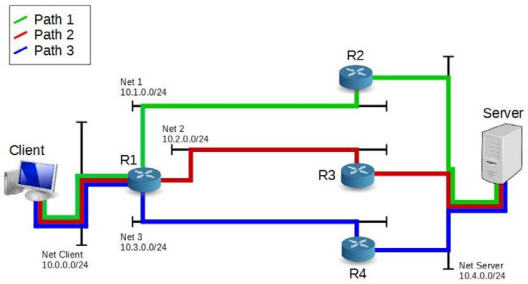

Fig. 4. QUIC multipath testbed

Delving a bit deeper into the details, each experiment consists in the client sending an HTTP/2 request for a page stored on the server and getting back the response. The objective of each run is to measure the time needed to complete the transaction, as a function of both the dimension of the requested page and the number of paths traversed. The latter parameter varies between 1 and 3, whereas the former falls in a range between $500KB$ and $2MB$, with a step of $500KB$. We have run the experiments 35 times for each configuration. The average value of each set of repetitions has then been used to draw the comparative analysis described in the following.

For the first set of experiments, we try to define our benchmark, by relying on the original code base and hence leveraging a single communication path. The server listens to incoming connections on port 6121, while the client creates a UDP socket associated with port 8001, connects to the server and sends a request for the desired page. In such a case we have a single path, identified by the pair of addresses <10.0.0.1:8001, 10.4.0.4:6121>.

For both the second and the third set of trials, we instead leverage our modified code base in order to perform multipath-enabled transmissions. We have either two or three parallel paths that get exploited in a round-robin fashion. The average execution times (in milliseconds) are reported in Table I, as well as graphically reproduced in Fig. 5

TABLE I. Measured average execution times

# of Paths \ Page size	500KB	1MB	1,5B	2MB
1	48,31	93,68	123,83	182,34
2	68,80	125,60	182,03	319,77
3	71,34	128,06	194,31	406,60

As expected, multipath transmission introduces an overhead which grows as the dimension of the requested page increases. Indeed, such an increase in page dimension entails that the number of exchanged packets increases as well. We have actually identified a non-optimal management of acknowledgments, timers and retransmissions as the most probable culprits

behind the detected performance degradation. Needless to say, the presence of multiple alternative paths, all of which can be exploited at the same time and with dynamically changing configurations during the same session, unavoidably makes time-dependent communication management much more complicated when compared to the single path case. Though, as stated above, the undeniable improvements in terms of reliability, robustness and security (not even to mention things like traffic offloading in the presence of alternative network connections to the Internet which might act as bottleneck links) definitely play in favor of multipath.

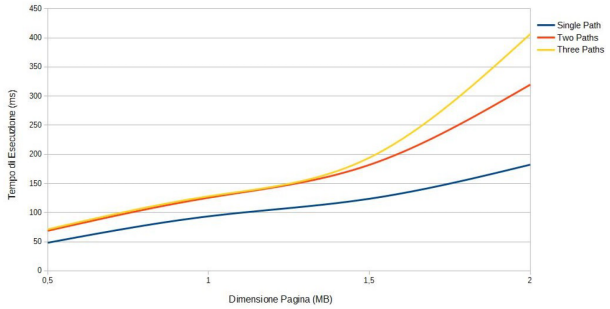

Fig. 5. Measured average execution times

VI. Lessons learnt and Future Work

In this paper we have presented all of the phases that have brought us to the introduction of the multipath functionality 'inside' the QUIC protocol. We use the word *inside* on purpose, since our approach has gambled on the assumption that QUIC (which is still a work in progress as concerns standardization) would bend towards native support for multipath. This was indeed the direction of the standardization work at the time when we started our design and implementation efforts.

As a matter of fact, the IETF community has lately taken a different direction and seems to have reached consensus on delaying multipath work to riper times. Multipath has hence gone out of the group's short-term sights and is currently looked at as one of the first protocol extensions to be worked upon once the basic functionality of the protocol has been successfully completed. The extension approach has actually been recently undertaken by the authors of the work in [6], who designed a QUIC extension enabling a QUIC connection to use different paths such as WiFi and LTE on smartphones, or IPv4 and IPv6 on dual-stack hosts. Rather than starting from the Google Chromium code base, the authors of the work in question implemented their proposal as an extension to the quic-go implementation [7].

Coming back to the future of multipath in QUIC, by carefully reading through the QUIC mailing list archives, it looks like such a capability is definitely out of the so-called v1 version of the protocol. At the time of this writing, the related initial milestones are being shifted and it seems most likely that adoption by the working group will probably not happen before December 2019, with completion indicatively scheduled around May 2020. We nonetheless believe that the work herein proposed does represent a useful piece of research, since it allowed us to work on a clean design of the multipath functionality, as well as to derive interesting data from the real-world implementation we produced. The preliminary results we obtained, while susceptible of substantial improvements, look definitely promising and motivate us in keeping on working on this subject. The work we have done until now[2] can indeed be considered as a good starting point. From there, we will first of all have to redesign some of the mechanisms that we proposed, so to let them adhere to the above mentioned changes in the overall QUIC structure, which sees multipath as an extension rather than an intrinsic feature of the protocol. Once done with this, we will have to work on the introduction of new packet and frame types for the creation, addition and removal of paths. We will also design a mechanism for the effective management of the multiple layers of packet numbers, so to allow multiplexing both at the stream- and at the path-level. This would enable packet retransmissions along alternative paths. A good deal of efforts should also be devoted to improving retransmissions through advanced, multipath-aware, congestion-control and loss-detection techniques. Such techniques should leverage both path-level and connection-level control mechanisms. With such fine-grained controls we might, e.g., identify lossy paths and divert part of the traffic (retransmissions included) along less congested alternative routes characterized by lower error rates. Finally, a lot of space for improvements derives from the identification of multipath transmissions scheduling techniques going beyond the naif round-robin mechanism we implemented as a proof of concept. Such techniques might range from the parallel transmission of data packets along multiple paths with the aim of bandwidth aggregation, to the possibility of dynamically activating alternative communication links in case of failures or performance degradation along the selected primary path.

Acknowledgements

This work was partially funded by ESA (European Space Agency), within the framework of project VIBES, ESA Contract N.: 4000122991/18/UK/ND.

References

[1] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-quic-transport-17, Dec. 2018.

[2] M. Bishop, "Hypertext Transfer Protocol (HTTP) over QUIC," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-quic-http-17, Dec. 2018.

[3] M. Thomson and S. Turner, "Using TLS to Secure QUIC," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-quic-tls-17, Dec. 2018.

[4] J. Iyengar and I. Swett, "QUIC Loss Detection and Congestion Control," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-quic-recovery-17, Dec. 2018.

[5] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk, "QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2," Working Draft, IETF Secretariat, Internet-Draft draft-hamilton-early-deployment-quic-00, Jul. 2016.

[6] Q. De Coninck and O. Bonaventure, "Multipath quic: Design and evaluation," in *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '17. New York, NY, USA: ACM, 2017, pp. 160–166. [Online]. Available: http://doi.acm.org/10.1145/3143361.3143370

[7] L. Clemente, "A QUIC implementation in pure go," Github, Open Source Project, Project repository: github.com/lucas-clemente/quic-go.

[2] All of the materials, including source code, of our project are publicly available at the following gitlab repository: http://gitlab.comics.unina.it/AT-Thesis/QUIC_Multipath/