IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 13, 2021, accepted November 16, 2021, date of publication December 23, 2021, date of current version January 18, 2022.

Digital Object Identifier 10.1109/ACCESS.2021.3137991

Programmable Session Layer MULTI-Connectivity

SEPPO HÄTÖNEN, ASHWIN RAO, AND SASU TARKOMA, (Senior Member, IEEE)
Department of Computer Science, University of Helsinki, 00014 Helsinki, Finland

Corresponding author: Seppo Hätönen (seppo.hatonen@helsinki.fi)

This work was supported in part by the Academy of Finland IDEA-MILL under Grant 334934, and in part by the Business Finland 5G FORCE Project.

ABSTRACT Our devices can use a wide range of communication technologies such as multiple cellular technologies (4G/5G), WiFi, and also Ethernet. At the same time, applications have a choice of a wide range of transport protocols such as QUIC and TCP that can be fine-tuned and optimized according to their needs. However, in spite of these advances, offering seamless multiconnectivity to applications continues to be a hard problem. The key factors that continue to be a roadblock towards achieving seamless multiconnectivity include a) applications cannot specify the communication technologies to be used by their flows, and b) the traditional definition of a connection endpoint was not designed to support mobile nodes. In this paper we discuss the key challenges that make this problem hard. We also present MULTI, a session layer approach that can be leveraged to address some of the key sub-problems of this problem. For instance, we observe that MULTI incurred a small overhead (less than 5% decrease in throughput) when using TCP compared to the native asyncio python library.

INDEX TERMS Multiconnectivity, programmable, session layer, asyncio, QUIC, TCP.

I. INTRODUCTION

Our mobile devices, including laptops and smart phones, can use multiple different communication technologies and multiple transport protocols. However, utilizing these technologies and protocols to their full potential continues to be difficult and off-the-shelf mobile devices still continue to use a single communication technology for data transfer. For instance, Android devices give WiFi a higher priority than cellular due to different reasons such as monetary, bandwidth or latency [1].

Some of the key reasons that stop us from efficiently utilizing these technologies and protocols are as follows.

1. Applications cannot easily specify the communication technology that should be used for its flows. Different applications have different requirements for their connectivity. Some may require bandwidth, while some require low latency. Devices typically allow using only a single communication technology for data transfer, and the applications have almost no ability to specify their preferred choice of communication technology. For instance, our smart phones typically give WiFi a higher priority than cellular [1]. On Linux devices, the priority of communication interfaces is typically device-wide, and applications require specific capabilities or superuser privileges to specify application specific interface priorities [2]. Similarly, versions of Android may allow restricting applications to either mobile networks or WiFi through system settings or through a firewall application [3], [4].

2. The traditional definition of a connection endpoint was not designed to support mobile nodes. An UDP or a TCP transport connection is defined as a five tuple – source IP address, source transport port, destination IP address, destination transport port, and transport protocol. These five values are present in the packets using the connection, and the four values other than the transport protocol may be modified at different in-network functions in the path between the source and the destination. Consider a packet originating from a mobile device to a remote server. For example, network address translators can modify the source IP address and the source port number while reverse proxies may modify the destination IP address and port numbers [5], [6]. Clearly, the definition of a transport connection is not truly end-to-end because the same connection may be defined by a different five tuple at the source and the destination [7], [8].

Over the years, there have been multiple different solutions that have been proposed to address these issues (see §II).

The associate editor coordinating the review of this manuscript and approving it for publication was P. Venkata Krishna.

However, Internet ossification makes it difficult to use the proposed protocols in the wild [9]. Regardless, what is common to all of them is that they either tackle a specific use case or require special infrastructure and software. For instance, the shortcomings of using the five tuple as a connection endpoint requires mechanisms that are agnostic to the changes at underlying transport and network layer. [1], [10], [12]. QUIC and its extension MultiPath QUIC (MPQUIC) exemplify such a mechanism [1], [10], [12]. They are designed from the beginning to be transport protocols over UDP that are agnostic to the source address and port by using a Connection ID inside the packets. The Connection ID allows the QUIC server to associate packets with same Connection ID to an established connection regardless of the source of the packets. Similarly, Multipath TCP (MPTCP) [13] uses special TCP headers to carry a Connection ID that allows end hosts to combine packets with a different five tuple to the same connection. Another example is Mosh [14], which addresses this issue with its State Synchronizing Protocol (SSP); SSP can be broadly categorized as a session layer protocol that uses the services of transport layers.

In this article, we first discuss some of the key challenges that continue to make seamless multiconnectivity hard. We then present an example of a programmable session layer solution, MULTI, that is aimed at addressing some key sub-problems of this problem. Among the key goals of MULTI is the ability to allow applications to specify the preference of protocols, and also suggest configurations for the protocols. MULTI acts as a shim at the session layer that also allows multiplexing and aggregating data over multiple connections. Using multiple, potentially different transports allows MULTI to achieve connectivity over different networks with different policies. As establishing connectivity and multiplexing data over connections is performed at the session layer, MULTI can be deployed rapidly and allows fast deployment of new features, including more advanced schedulers that determine how, for example, link aggregation is handled.

Our key contributions are as follows.

1) Our solution, MULTI, attempts to work around Internet ossification and provides an umbrella which incorporates past approaches. MULTI combines the best of previous works in multiconnectivity and builds on their insights.
2) We detail the characteristics of MULTI which allows applications to simultaneously use multiple transport protocols and interfaces with different characteristics. Specifically, MULTI allows applications to request connections with certain characteristics including configurations for the transport and Internet protocol, the interfaces and interface configurations, etc., and this in turn enables MULTI to multiplex data over multiple transport layer connections.
3) We provide an open source proof-of-concept implementation of MULTI for evaluation purposes. We believe that our prototype of MULTI can be easily extended to include features such as custom schedulers for multiplexing data over multiple connections and also supporting new transport protocols.

MULTI builds on the insights of Mobile IP [15], QUIC, and MPTCP. We believe that it is the next step in the series of works that have been aimed at offering mobility and multiconnectivity. Specifically, Mobile IP allows the interfaces and the networks to change, but the IP address assigned to the device stays fixed. Using QUIC allows applications running on mobile devices to remain connected when the used IP address and interface changes by using a Connection ID, while MPTCP allows applications to multiplex data streams over multiple interfaces. In contrast, MULTI allows applications to multiplex data stream over multiple transport protocols, each of which can use different IP addresses and interfaces.

We implement MULTI using state-of-the-art asyncio primitives of Python [16], and the asyncio QUIC library [17]. We show that it can support more than one transport protocol, and more than one link layer technology. We also show that it can achieve a throughput and latency that is comparable to the protocol implementations it uses. For instance, we observe that MULTI incurred a small overhead (less than 5% decrease in throughput) compared to the base line asyncio when using TCP.

II. BACKGROUND

There have been many different multiconnectivity solutions developed over the years. These solutions operate in different levels of the network stack, such as network, transport, and application layers. Some of the solutions also operate in multiple layers, as they may use capabilities of a layer above or below the actual layer where they operate. The solutions can also be roughly divided into those that do multihoming [15], multipath [12], [13], and those which do both to various degrees [18].

In this section, we briefly describe several of the solutions and categorize them based on their characteristics. We also discuss why multiconnectivity is still hard in the current Internet.

A. NETWORK LAYER AND BELOW

In this section, we go through several multiconnectivity methods that belong to the network layer. These solutions operate under the transport layer and aim to provide either multihoming or multipathing. The Table 1 shows the main differences of network layer protocols discussed below.

1) HIP

Host Identity Protocol (HIP) is a technology that separates the endpoint identifier and locator roles of an IP address, and creating a new name space that allows mobility and multihoming [19], [21]. HIP is implemented as a layer between the transport layer and the IP layer, creating a new Host Identity Layer between them. For identifying hosts, HIP uses cryptographic Host Identity Tags, which are exchanged between

TABLE 1. Characteristics of different multihoming or multipath network layer protocols. *Each column highlights the main differences of each protocol. For example, HIP requires both endpoints to support HIP, while support for Mobile IP and SDN Mobility is only required at the mobile endpoint.*

Protocol	Home Network	Rendezvous Server	Protocol Stack Changes	Endpoint Support
HIP [19]		✓	✓	Both endpoints
Mobile IP [15]	✓		✓	On mobile endpoint
SDN Mobility [20]	✓			On mobile endpoint

hosts and rendezvous server. Every time a host moves into a new location, i.e. its IP address changes, the host updates the rendezvous server of its new IP address.

When two hosts need to communicate with each other, they first contact the rendezvous server through for example the DNS system. The server responds with the current location of the destination host, allowing hosts to communicate.

While HIP allows mobility and multihoming, there are drawbacks. First, both hosts need to support HIP in their operating system. Second, there need to be special rendezvous servers in the Internet with static locations. These require resources to run, and are susceptible to attacks.

2) MOBILE IP

Mobile IP is another way to allow multihoming and mobility [15]. In Mobile IP, each host has its own Home Network. Withing the Home Network, the host has a permanent IP address. When the host is away from the home network, it receives a so-called Care-of Address from the foreign network it is attached to at that time. The host then registers the Care-of Address with a Home Agent inside the Home Network. This allows the Home Agent to tunnel traffic to the host's permanent address to the host's current Care-of Address, and allows applications to remain unchanged.

As with other systems, Mobile IP does have its drawbacks. Main drawback is the need for the Home Network, which needs hosting resources. Since traffic is relayed through the Home Network, it also causes extra latency, and depending how Home Network is connected to the Internet, possible bandwidth caps.

3) SDN-BASED MULTIHOMING

On-device virtual switches managed by Software-Defined Network (SDN) controllers can also be used to offer multiconnectivity. Meghna [20] is one such example for network-driven multihoming. Meghna uses an SDN switch on the host device that is connected through different interfaces to an SDN capable network. The host switch, or more precisely the traffic managed by the host switch, is controlled by a network SDN controller. The host switch bridges all network interfaces together, and exposes a single interface to the applications. The traffic is then forwarded to the network using the interface selected by the controller.

The limitation of this approach is that it requires a home network. If the device is roaming away from the network, the Meghna establishes a VPN connection to the home network, and the traffic is first forwarded to the home network and then beyond. This causes overheads as the traffic has to take a longer non-optimal route. On the other hand, the Meghna allows the applications to retain the same IP address independent to their physical network location, while the main drawback is the requirement of the home network.

B. TRANSPORT LAYER

In this section, we discuss several transport layer technologies. What sets them apart from the lower network layer is that they aim to provide their capabilities over either extending existing transport protocols such as TCP with multiconnectivity features, or provide new transports that have been designed with multiconnectivity in mind. The differences between the discussed protocols are shown in Table 2 for reference, and the details are discussed below.

1) MULTIPATH TCP

Multipath TCP (MPTCP) [13] extends the regular TCP protocol. It was developed to add multipath support to TCP, i.e, allow the simultaneous usage of all available network addresses for a TCP flow by creating sub-flows over the addresses. MPTCP can aggregate all available links for bandwidth, perform seamless migration, and choose a lowest latency link for interactive usage.

MPTCP is implemented by special TCP options. An MPTCP host announces that it supports MPTCP on opening a TCP connection. If the destination is also MPTCP capable, they negotiate the connection, announce their other addresses, and initiate sub-flow establishment over other addresses. If a sub-flow becomes invalid, for example due to roaming between networks, the host announces that the particular subflow is invalid and should be removed.

Allowing MPTCP to use all possible interfaces increases the available bandwidth, but can incur extra monetary costs over cellular links. In addition, using simple path managers can allow the traffic to also traverse networks that are undesirable due to various reasons such as security. However, MPTCP supports more advanced path managers that can be used to achieve desired interface usage based on existing policies or user input.

2) QUIC

QUIC [1], [10], [23] is a transport protocol built over UDP to facilitate better performance for HTTPS. QUIC is designed to multiplex multiple data streams into a single QUIC connection as many web pages contain multiple small elements, and opening a new connection for each of them is expensive.

TABLE 2. Characteristics of different multihoming and multipath protocols. *Each column highlights different aspects of the protocols and where they differ.*

Protocol	Multipath	Multihoming	Connection Endpoint Identifiers	Implemented in Userspace
MPTCP [13]	✓	✓	Set of IP addresses and port numbers	
QUIC [10]			Connection ID	✓
MPQUIC [12]	✓	✓	Connection ID and set of IP addresses and port numbers	✓
SCTP [22]		✓	Set of IP addresses and port numbers	

Unlike TCP and MPTCP, QUIC is implemented in userspace, and this allows easier deployment as it does not require modifications to the OS kernel. QUIC supports mobility by including a Connection ID in the QUIC headers. This Connection ID allows QUIC to resume connections that would otherwise be broken on IP address and port changes, for example due to switching from WiFi to cellular network.

Although QUIC is designed for HTTPS, applications can use it as a transport protocol to exchange other data streams.

3) MPQUIC

Even though regular QUIC is not dependent on the classical 5-tuple describing connections due to the Connection ID, QUIC does not support multipath connections. Currently, there is a proposal to extend QUIC with multi-patch capabilities known as Multipath QUIC (MPQUIC) [12].

MPQUIC is an extension to QUIC. Its operations are similar to MPTCP. When a QUIC connection is established, the peers of the connection exchange their IP addresses, and check if they can exchange flows between themselves over the new connections. Similarly to MPTCP, MPQUIC can tolerate a loss of connections, and use the rest of the flows as before.

4) STREAM CONTROL TRANSMISSION PROTOCOL

Stream Control Transmission Protocol (SCTP) is a protocol designed for reliably transferring messages between endpoints over UDP [22]. Unlike TCP, SCTP transfers messages instead of constant stream of data over UDP. The messages are encapsulated in chunks that are transferred inside SCTP packets.

SCTP provides congestion control and supports multihoming. Both endpoints of a SCTP connection can have multiple IP addresses, and connectivity between them is probed during the connection establishment. If any of the available connections fail, the rest of the connections can still be used.

While the SCTP was published in 2000, it has been plagued by the lack of support in middleboxes and lack of awareness. As such, while SCTP would provide many of the features desired in multiconnectivity, it cannot be relied as the only option.

C. APPLICATION LAYER

Here we discuss two approaches for multiconnectivity in the application layer. Namely, these methods do not rely on underlying layers, but embed relevant information in the application data to allow hosts to move.

1) MOSH

Mobile Shell (MOSH) [14] is a remote terminal application that can handle roaming between networks. It serves as an example of a session layer approach to offer multiconnectivity. MOSH is primarily aimed at addressing the problems of SSH, including no roaming and sleep. These problems largely stem from the way SSH connects to the server and transfers data. The connection is tied to the 5 tuple describing the connection, and the data is transferred as a continuous data stream, i.e. all bytes need to be transferred and shown in order.

MOSH does not transfer data in a stream, instead it synchronizes objects. Consequently, a MOSH user sees the latest visible terminal data instead of having to go through all the backlog of the terminal.

MOSH operates over UDP that allows datagrams to be sent over a UDP socket regardless of the current IP address. Like QUIC, MOSH also uses connection identifiers embedded in the UDP datagrams with its State Synchronizing Protocol (SSP). This allow MOSH server to associate datagrams from different clients to specific sessions, allowing MOSH to achieve roaming support. SSP can be broadly categorized as a session layer protocol that uses the services of transport layers. However, as MOSH is only a remote terminal application, MOSH cannot be used as a transport protocol.

2) BUFFERING

Another way to handle multiconnectivity is buffering data. This approach is applicable to streaming video or similar from the network, where there is no realtime component to data, i.e. the stream is predetermined and does not have changing elements [24]. In this approach, the video streaming application uses cookies or similar to carry the Connection ID and buffers the received data before showing it to the user. If the network connectivity changes, the application still has buffered video to show while it tries to reconnect to the service. At best, the user does not even realize the network has changed as long as the reconnect happens before the buffered video runs out.

This approach is not applicable to any use cases, where the content changes or there is any realtime component, such as gaming in the extreme case or even browsing web pages. In these cases, the user expects to receive the data as fast as possible with minimal buffering.

D. TAPS

Earlier examples of multiconnectivity protocols discussed above are single technologies that solve multiconnectivity

in their own niche. However, they are not able to provide a holistic approach to multiconnectivity. The Internet Engineering Task Force (IETF) Transport Services (TAPS) working group (WG) is working on defining an architecture for exposing a Transport Services API to the application developers [25]. The goal of the Transport Services API is to allow application developers easier access to transport protocol services such as multiple IP addresses, multipathing, and providing multiple application streams.

Traditionally the socket API provides access to different transport protocols such as TCP and UDP. However, different protocols have different methods for accessing them and are not used consistently. In some cases, conceptually similar protocols, for example TCP and Transmission Layer Security (TLS) that both provide reliable data streaming services, use different calls to access the send and receive services of the protocols. Similarly, different protocols use different terminologies for the same concepts such as connection, flow, and messages. These create a burden for application developers to learn the differences of each transport protocols, including the calls to be used and the terminology. This has caused a stagnation on what protocols are actively used in the Internet, namely TCP and UDP.

The TAPS WG has also identified a set of services that different transport protocols offer, including services that can be handled automatically by the operating system, and services that require interaction from the applications. This identification has allowed TAPS to specify a minimal set of required services that the Transport Services API needs to provide to the application.

The Transport Services API defines the mechanism for applications to create network connections and perform data transfer. The API is an asynchronous, event driven system that uses messages to transfer data. Each call is designed to be asynchronous, i.e. the calls do not block the application.

E. NEAT

NEAT is a framework for platform and protocol independent transport API [26]. It is a userspace framework that allows application developers to use different transport protocols with minimal interaction from the application side. The NEAT also allows applications to specify different options and requests to the operating system on what transports should be used and how they should be configured. As such, NEAT can be considered a prototype implementation of TAPS.

The NEAT framework consists of the NEAT user module, which includes the NEAT API that applications use to request a connection. The NEAT module's policy manager and other components handle gathering candidate connections based on policies and cached information.

The NEAT framework performs Happy Eyeballs connection candidate gathering for each available or requested protocol [27]. For example, if the transport policies advocate SCTP or TCP, the NEAT performs connectivity checks to determine what IP address (IPv4 or IPv6) and which protocols work. This allows NEAT to discard non-working protocols and cache working protocols for future use.

The NEAT framework provides the best transport solution based on the request the application makes to NEAT, the policies defined by the system and its administrators, and the transports the network offers. This allows NEAT to provide the best single transport solution to the application.

The goals of TAPS and NEAT are very similar to the goals of MULTI which are discussed in §III-A. The main difference between NEAT and MULTI is that NEAT aims to provide the best single transport solution such as MPTCP or QUIC, MULTI aims to provide multiple transports that can be used simultaneously.

F. WHY IS MULTICONNECTIVITY STILL HARD?

Solutions described in this section try to solve multiconnectivity in different ways at different layers. Some of them do it at the network layer, some at session, at the application, or mixing them. Unfortunately, each of them have their own niches and requirements from the hosts, servers, networks, and middleboxes.

1) FIVE TUPLE CONNECTION ENDPOINTS DO NOT SUPPORT MOBILITY

The transport and network protocols used today were primarily designed to offer connectivity to devices that were either placed in static locations or did not move from one network to another. The end-to-end principle [28] is also largely violated with the introduction of load balancers, network address translators (NAT), and in-network middleboxes that modify the packet headers and payloads [9]. As a consequence, the five-tuple definition of a connection endpoint is no longer valid in a large number of networks.

2) MIDDLEBOXES MAKE IT DIFFICULT

Middleboxes and firewalls also ossify the protocols used because they tend to drop packets from protocols that are not well-known. For instance, firewalls blocking UDP datagrams make it difficult to introduce new UDP-based protocols or provide extensions to existing UDP-based protocols such as QUIC [1], [10], [12].

Stateful middleboxes that maintain the state of the connection also hinder multiconnectivity. When a device changes its location, the 5-tuple becomes obsolete, and causes the connection to break unless the middlebox can somehow tie the old and the new location together. Furthermore, encrypting the payloads of the packets aggravates this problem.

3) DEVICES MAKE MULTICONNECTIVITY HARD

Another thing that makes the multiconnectivity hard are the devices themselves. The internal routing of the devices are driven by predefined rules such as an Ethernet interface will have higher priority than a WiFi which in turn has a higher priority than a cellular interface, even though the application could reach the destination using any of them. These rules do not take into account the current network state. For example,

the WiFi interface of the device could be connected to a network with very slow uplink, while the cellular network could have a much faster uplink. A user (or application) cannot easily specify what communication interfaces to use, apart from manually turning an interface off and forcing the device to use other interfaces.

One approach to perform system-level traffic steering is to set routing table rules. However, these are system-wide changes and are hard to do per application basis, and they can break connectivity completely by accident. Furthermore, applications require super-user privileges to perform these changes.

To summarise, currently there is no single solution that would solve the issues of multiconnectivity. Each solution handles a specific use case or area, requiring specific infrastructure and software support.

III. OUR SOLUTION: MULTI

We design MULTI to complement the end-to-end principle of system design, in which applications draw a modular boundary around the communication subsystem and define an interface between it and the rest of the application [28]. In the previous section, we highlighted that TAPS and NEAT exemplify the design of such an interface. In this section we detail how MULTI builds on the insights of TAPS and NEAT to multiplex data over multiple transport layer connections, potentially using different protocols simultaneously. MULTI enables application developers to support and also adapt to the protocols in the link layer, network layer, and transport layer that are available on the networks to which devices using MULTI are connected. Specifically, MULTI acts as an umbrella for different solutions that have been developed over the years and it enables applications to make requests and provide hints and intents on its expectations and demands.

In §III-A, we discuss our goals, followed by architecture design in §III-B. Finally, in §III-C we discuss our implementation details of the proof-of-concept version of MULTI.

A. GOALS

MULTI is designed to achieve the following goals.

1) END-TO-END EXCHANGE OF DATA STREAMS

Our aim is to allow applications to exchange streams of data where the order in which the data arrives matters; currently we do not focus on datagrams where data can arrive out of order. We envision MULTI to be used by an end-user device for exchanging streams of data with a remote host. We believe that if our solution can support exchange of streams data, it can easily be modified to support datagrams and also short messages. Currently, we limit MULTI to transport protocols that have native support for exchanging data streams. For instance, MULTI can use UDP-based transport protocols such as QUIC that have native support for exchanging streams of data, but it cannot use UDP directly.

2) SUPPORT DIFFERENT TRANSPORT, NETWORK, AND LINK-LAYER PROTOCOLS

We aim to offer applications the flexibility to choose from a wide range of transport, network, and link-layer protocols. We assume that devices that use MULTI will be able to avail themselves of the services of different networks, each of which supports a different set of network and transport protocols. Inspired by TAPS and NEAT, our goal is to enable applications to use the best protocols for each situation depending on the requirements, the current policies, and the capabilities of the networks available for the data exchange.

3) SIMULTANEOUSLY USE MULTIPLE TRANSPORT PROTOCOLS

Simultaneously using multiple different transport protocols allows MULTI to achieve connectivity over multiple heterogeneous networks. Networks use middleboxes [29], and Honda et al. [9] highlight their impact on the ossification of the Internet. As a consequence, the set of transport protocols supported by the networks to which a device may connect cannot be known a priori. For instance, some networks may restrict or disallow usage of protocols such as MPTCP due to its special TCP headers or have policies against UDP flows that in turn affect QUIC. Unlike TAPS and NEAT which attempt to identify which transport protocols may be supported by the networks, and select one of the supported transport protocols, we believe that applications can benefit with the ability to multiplex data streams over multiple transport layer connections. Using multiple different protocols over the networks allows MULTI to achieve connectivity when a single protocol could fail. We believe that the idea of aggregating across multiple transport protocols makes MULTI a novel approach to offer seamless multiconnectivity.

4) ALLOW APPLICATIONS TO a) SPECIFY THE PREFERENCE OF PROTOCOLS, AND b) SUGGEST CONFIGURATIONS FOR THE PROTOCOLS

Along with allowing applications to be agnostic to the underlying protocols, we also aim to support a fine-grained control of the protocols. This is essential to ensure that flexibility does not come at a cost of reduced control over the underlying protocols used. Preference of protocols is vital because some protocols may be optimal for the application, but at the same time applications would prefer to fall back to alternative protocols if the optimal ones are not supported by the networks. For instance, an application might prefer QUIC over TCP when both MultiPath QUIC and MultiPath TCP are not supported by the networks.

5) SEAMLESSLY REACT TO NETWORK CHANGES

One of our aims is to support seamless multiconnectivity. In modern networks, devices regularly move between different networks, usually between WiFi and mobile networks. When the devices roam between these networks, the IP addresses of the network interfaces change or the

connectivity is broken. To allow MULTI to seamlessly react to these changes, MULTI needs to be able to detect network changes, and either automatically switch to available connections or resume connection when network connectivity has been re-established.

6) USERSPACE IMPLEMENTATION

Inspired by QUIC, MULTI is designed to be implemented in the user space. Furthermore, unlike TAPS and NEAT, we assume that MULTI will be supported by both endpoints. We make this choice because userspace implementations allow for faster deployment.

In the rest of this section we detail our approach to achieve these goals.

B. ARCHITECTURE

As shown in Figure 1, MULTI allows applications to exchange bi-directional data streams. It takes as input a bi-directional stream of data, and multiplexes it over multiple transport, network, and link layer protocols. MULTI bundles multiple solutions such as QUIC and MPTCP under one roof because it is aimed to harness the strengths of previous attempts to offer seamless multiconnectivity. It is also designed to be implemented in user space to allow easy deployment.

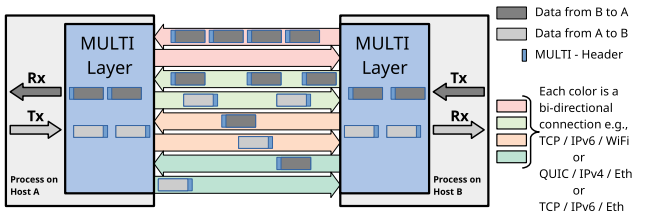


FIGURE 1. Example MULTI Session. *Processes can use MULTI to exchange bi-directional data streams. MULTI multiplexes the data over multiple bi-directional transport connections.*

In the following paragraphs we detail how MULTI allows applications to multiplex data stream over multiple transport protocols.

1) SESSION LAYER ENDPOINTS

MULTI requires both the source and destination of a data stream to support it. It is implemented above the transport layer of the protocol stack, specifically at the session layer. A MULTI session is uniquely identified by an application using a Session ID. This Session ID can be either specified by the application or generated at run time during the session initiation.

2) INITIATING A SESSION

A host (initiator) can initiate a MULTI session by opening a transport layer connection with a remote host. After the transport layer connection is established, the initiator provides the Session ID to uniquely identify a MULTI session. Once a session is created, MULTI uses this Session ID when opening subsequent transport layer connections that correspond to this session. After a session is initiated, the two endpoints exchange data using MULTI segments.

TABLE 3. Fields in the MULTI header for each MULTI segment.

Field	Description
Version	Version number
Segment type	Indicates the type of segment. Our prototype for MULTI currently supports segments for data, acknowledgement, keep-alives, and for closing a multi session.
Header length	The length of the header. This is used to indicate when the payload begins.
Session ID	The session id to which the segment belongs.
Sequence Number	Contains the sequence number if the segment contains data.
Segment length	The total length of the segment including the segment header.

3) MULTI SEGMENTS

As shown in Figure 1, MULTI segments are encapsulated within the payload of the transport-layer protocols. Consequently, the MULTI segments may be encrypted when using transport protocols such as QUIC that encrypt the payload.

Each MULTI segment includes a header whose fields are presented in Table 3. We would like to point out that these fields were chosen only to demonstrate the benefits of multiplexing transport-layer connections, and have not been optimized to improve MULTI's performance. Furthermore, because MULTI is implemented in userspace, we envision that these fields can be updated, modified, and optimized by the applications that use MULTI. To allow potentially different versions of MULTI to be able to establish connections, we include a *version* field in the header. This field allows newer version of MULTI to be compatible with older versions of MULTI. Although having a version in the header can expose some vulnerability for instance in the TLS [30], it can be mitigated by negotiating the suitable non-vulnerable versions.

A key field in the header is the Session ID. As previously mentioned, the same Session ID is used across the transport-layer connections of a given session.

MULTI segments can either be control plane segments, or contain the data. Control plane segments include keep-alive messages and a special segment to explicitly close the session by closing all the open transport-layer connections of that session. The data plane segments include segments that encapsulate the data, and the acknowledgments to identify that the segment was successfully received.

4) MULTI CONTROL PLANE

Before opening a MULTI session, the application needs to provide the configuration that MULTI can use for transporting the data to the remote host. We provide an example configuration in Figure 2. The design is motivated by

```
{'connection_priority': [('mptcp','ipv6',['eth',
↪  'wifi']),
  ('quic','ipv6', 'cellular'),
  ↪  ('tcp','ipv6','cellular')],
'multi_config': {'connect':'seq',
↪  'scheduler':'RR'},
'remote_host' : 'mcserver',
'session_id'  : 'random',
'mptcp_config': {'scheduler':'RR'},
'quic_config' : {'idle_timeout':5},
'tcp_config'  : {'nagle':false,
↪  'idle_timeout':10},
'ssl_config'  : {'cert':'cert_name',
↪  'tls':'TLS_v3'},
'wifi_config' : {'power_savings':false}}
```

FIGURE 2. **Example MULTI Configuration.** ***Applications can specify the priority for the protocols along with the configuration for each protocol.***

URLSession [31], and it is designed to be verbose enough to meet our goals.

The first entry is the `configuration_priority`, i.e., the suggested order for the transport-layer connections. Each connection is defined by a triple: the transport-layer protocol, the IP-layer protocol, and the interfaces that can be used for the connection. Applications using MULTI can use the connection priority to explicitly specify their preferred preference of the protocols from the protocol stack. In the example shown in Figure 2, the first entry for `connection_priority` specifies that MULTI can use MPTCP, and this MPTCP connection can use IPv6 and the Ethernet and WiFi interfaces; this allows the application to specify that it is optimized for MPTCP, but it might prefer QUIC over TCP when MultiPath TCP is not supported by the network. MULTI opens connections similar to Happy Eyeballs (HE) [27]. In HE, multiple connection establishment attempts are launched simultaneously over available transport interfaces and protocols; when a connection is available, it is given to the application while the rest of the finished connections are cached for future use.

Similar to HE, the suggested order for connections, along with the fields for `multi_config` allows MULTI to determine how the connections are created and used. For instance, the `connect` field of `multi_config` shows that the connections are configured to be opened sequentially.

Connections can be opened sequentially or in parallel. Opening a connection in parallel is useful for minimising latency before the first data byte is transferred. As soon as the first connection is open, MULTI can use that for data transfer, and when the rest of the connection establishment attempts finish, they are added to the list of usable connections. When there are more than one connection open, MULTI can start using them as specified in the configuration.

In contrast, sequential trying to open connections can increase the latency before the first data byte is transferred, but can have other benefits. The main benefit of sequential opening is restricting the load induced to both the device and the destination system. As connection establishment attempts arrive sequentially, the destination system is capable of handling more unrelated connection attempts simultaneously. For the host, the sequential opening may even allow better power management, as network interfaces are woken up only when needed, and not all simultaneously. The main cause of latency when opening the connections sequentially are failed connection attempts. If a connection establishment fails, detecting it can induce extra time before moving to the next connection on the list.

The `scheduler` field of `multi_config` specifies how the data is multiplexed all the open connections. In the example presented, the application specifies that it would like MULTI to use the Round Robin scheduler to distribute the load evenly across the opened connections. The scheduler operates in user space and this opens avenues for creation of application specific schedulers. For instance, applications can specify lowest latency to indicate that the transport connection with the lowest latency is to be selected, or it can specify that the applications can use the *top n* connections that have the lowest latency, or they can select the *top n* connections with the best throughput. Furthermore, along with opening the connections sequentially or in parallel, MULTI can be easily modified to support staggered connection attempts.

However, MULTI is restricted to what the network can offer. For example, if an application requires low latency connection, MULTI can only offer what the available networks can provide. In this case, MULTI can be extended to use the connection with the lowest latency and share the measured latency across all connections with the application. The application can then decide whether to continue with the available latency or terminate the session.

The subsequent fields in the configuration allow the application to specify how the MULTI layer should set up and use the connections. This allows MULTI to configure the transport, network, and link-layer protocols. In the example shown in Figure 2, MPTCP is configured to use the Round Robin scheduler, QUIC is configured to have an idle timeout of 5 seconds, the nagle algorithm is disabled for TCP, and TCP's idle timeout is increased to 10 seconds. Furthermore, the SSL certificates to be used by QUIC and TCP are also specified. Along with configuring the transport protocols, MULTI can also request the host operating system to configure the link layer protocols. In this example, MULTI can request the host OS to disable the WiFi power savings.

5) MULTI DATA PLANE

After providing the configuration, the application can open a MULTI session to a remote host. On opening a session, the application is provided with two handles: one for transmitting data, and the other for receiving data.

As shown in Figure 1, MULTI buffers the data sent by the application in the session layer. In this layer, MULTI uses the specified scheduler to multiplex the data transfer over the opened connections. MULTI also splits the data into segments and adds the MULTI header to each segment. As shown in Table 3, the header includes the sequence number for ordering the data. Note that MULTI relies on the underlying transport

protocols to perform the congestion control, and ensuring the reliable transfer of the segments. Similar to data transmission, data received over these connections is ordered according to the sequence number and buffered till the application reads from the buffers.

All fields of the MULTI control plane segments and data plane segments including the MULTI headers are encapsulated in the payloads of the transport protocol segments. This implies that these payloads may be encrypted if the transport protocol encrypts its payload. For instance, MULTI segments including the MULTI headers encapsulated in QUIC payloads are encrypted. We discuss the implications of encrypting the MULTI header in §V.

C. IMPLEMENTATION

We implement our MULTI prototype in python, and we have made our code and the scripts used for evaluating its performance publicly available. Our prototype currently supports TCP and QUIC as the transport-layer protocols, and is also designed to support MPTCP. We use the native asyncio implementation for TCP, and the aioquic library for QUIC [17]. We use the asyncio libraries because they are designed to allow developers to build networking applications such as web-servers.

MULTI requires the ability to a) simultaneously open and use multiple transport layer connections, and b) explicitly provide socket options that reflect the requirements specified by the user. The default asyncio library for TCP and UDP offers these capabilities, however, they are currently not available for the aioquic library. We therefore added these features to the aioquic library in the following manner. First, we made the aioquic client *awaitable*. This enables the MULTI library to simultaneously open multiple connections and use them concurrently. Second, the current version of aioquic does not allow setting socket options. In its default state, aioquic uses the OS routing to send UDP datagrams over any of the available interfaces. We therefore modified the aioquic to allow binding a socket to a specific interface to restrict the UDP datagrams to that interface. Note that by doing so we have violated the *sans-I/O* design principle [32] governing the design of the asyncio protocol libraries for TCP and QUIC. The *sans-I/O* mandates that the library code does not perform network I/O, however we violate it because we want MULTI to offer the flexibility to specify the interfaces to be used. In order to offer this flexibility, MULTI must bind the sockets it uses to the interfaces specified in its configuration.

From the application point of view, when a session has been created, MULTI exposes a stream reader and stream writer to the application. The application can use these handles as it would use normal sockets to send and receive data. Under the hood, MULTI multiplexes the sending and receiving of data over the available transport connections.

IV. EVALUATION

In this section we present the results of experiments to evaluate our MULTI prototype.

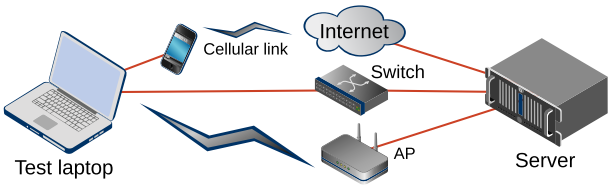


FIGURE 3. Testbed. *Our testbed contains our laptop that has an Ethernet, a WiFi, and a 5G connection to our test server.*

TABLE 4. Testbed components.

Component	Model / Version
Server	Intel Xeon E5540, Ubuntu 20.04 LTS
Laptop	HP Elitebook 820, Intel i5-5300U, Ubuntu 20.04 LTS
Access Point	Northbound Networks Zodiac WX
5G Phone	OnePlus 7 Pro
Python	3.8.5
QUIC	aioquic 0.9.9 (modified)

A. GOALS

The goal of our evaluation is to showcase both the strengths and weaknesses of the MULTI approach. Specifically, we aim to identify avenues to improve MULTI and highlight some of the factors that continue to make seamless connectivity hard, including overheads caused by a session layer approach and the effect of the packet scheduler when multiplexing multiple connections.

In our evaluation, we first focus on the throughput when transferring large amounts of data (256 MB), the duration to transfer a small amount of data (25 kB), and the time to open a MULTI session. We then show how MULTI behaves when the test device moves between networks, either aggregating or switching active connections. For this test, the client device initiates a download over a cellular connection and then joins a WiFi network. The details of our methodology for the above tests are presented in §IV-C.

B. EVALUATION TESTBED

For our evaluation, we use our testbed presented in Figure 3, and the hardware and software detailed in Table 4. Our client laptop can exchange data with our server via WiFi (802.11ac), a 1 Gbps Ethernet link, and an 5G mobile phone using USB tethering. Note that all the Ethernet links, including the link between the WiFi Access Point (AP) and our server, can operate at a bit rate of at least 1 Gbps. All link layer technologies, namely Ethernet, WiFi, and 5G, can use both IPv4 and IPv6 to reach the server. However, while the server can reach the laptop over all three technologies using IPv6, the server can reach the laptop only through WiFi and Ethernet when using IPv4. This is due to two levels of NAT between the laptop and the server over the 5G link: the ISP has IPv4 NAT between the phone and the Internet, and the USB tethering at the phone adds another NAT. Due to this, our baseline *iperf* tests

(described in §IV-C) are only run over IPv6 from server to the laptop.

The average round trip time (RTT) between the server and client over 5G, Ethernet and WiFi was 46 ms, 0.250 ms and 1.8 ms respectively. These RTT measurements were conducted using ping. Furthermore, we use a dedicated management network (not shown in the figure) to facilitate remote management, and also for all other background traffic traversing our laptop and server.

C. TEST DESCRIPTIONS

To evaluate our MULTI prototype, we first use three different test scenarios to establish the baseline performance of our testbed and also the performance of MULTI. Then, we use three tests to evaluate the current performance of our MULTI prototype, followed by two tests to demonstrate how MULTI handles roaming between networks.

In each test, the laptop is acting as client that connects to the test server. In each of the figures, the values shown in Rx (receiving) and Tx (transmitting) are from the laptop's point of view.

1) BASELINE PERFORMANCE

We measure the baseline performance for our testbed with the following three tests.

1. iPerf. We use iPerf to measure the achievable throughput when using our Ethernet and WiFi links for transferring data over TCP and UDP over IPv6. iPerf currently does not support QUIC, so we use UDP to give a rough estimate on the throughput that can be achieved when using UDP based transport protocols such as QUIC.

2. Base. For this test, we measure a) the time required to open a connection, b) the throughput for transferring 256 MB of data, and c) the total duration to transfer 25 kB of data, including the time to open the connections, when using the asyncio python libraries for TCP and QUIC. We run the tests over all links using TCP and QUIC over IPv6 in both directions.

3. Multi. We repeat the previous test using MULTI to quantify the bandwidth overheads incurred when it is used. Similar to the previous test, we configure MULTI to use only one transport-layer protocol (TCP or QUIC), IPv6, and one link layer interface.

2) MULTI PERFORMANCE

We consider MULTI with three connections; specifically, we open a transport-layer connection on each interface. As with the baseline performance, we measure a) the time to open a connection, b) the duration to transfer 25 kB of data, and c) the throughput when transferring 256 MB of data.

a: TIME TO OPEN A CONNECTION

A key component of MULTI is the module that allows it to open multiple connections for the data transfer. However, the time taken to open each of the connections can vary significantly due to link characteristics and also the configuration settings. For instance, MULTI can currently be configured to (i) open all connections sequentially and begin the data transfer after all the connections are open, or (ii) try to open all connections in parallel, and begin the data transfer after one of them is successfully opened. For this test, we consider option (i), i.e., sequential opening, because it is representative of the worst case scenario; the baseline measurements for MULTI are representative of option (ii), i.e., parallel opening.

b: DURATION TO TRANSFER 25 kB

Multipath protocols such as MPTCP can be inefficient when transferring small amounts of data as the time to open multiple connections can be larger than the time to transfer the data [33]. In this test, we emulate this scenario. Specifically, we perform the data transfer after all the connections have been established, and we use the Round Robin scheduler to distribute data evenly across the links. We use segment sizes of 16 kB, and allow the MULTI segments to be fragmented by the transport-layer protocols. For instance, the default aioquic implementation fragments each MULTI segment into segments of 1280 bytes.

c: THROUGHPUT WHEN TRANSFERRING 256 MB

In this test, we evaluate the performance of MULTI when using the Round Robin scheduler to distribute the load across the two communication links. Although the Round Robin scheduler is not the most efficient scheduler to maximize the throughput, we use it to showcase the multiplexing capabilities of MULTI.

3) MOVING BETWEEN NETWORKS

One of the main goals of MULTI is to support mobility. In previous tests, the networks the test laptop has been connected to have been static. However, when devices move between networks, the active connections are affected, and can cause degraded service.

Here, we emulate the laptop moving from one network to another, i.e. either switching between active interfaces or aggregating multiple interfaces when they become available. Aggregating multiple connections over multiple network interfaces should increase the total bandwidth available to MULTI.

Switching between networks can either cause total connection loss, or packet loss and momentary drops in bandwidth. We emulate network switching with make-before-break, i.e. the new connection has been established before the old connection is lost. In our tests, the network switch is triggered manually after 15 seconds. This emulates the case when the user has configured MULTI to use WiFi when available and cellular as a backup when WiFi is not available.

In real networks, the changes in networks happen when the devices move from one place to an another, and network changes can be very abrupt. Our current MULTI prototype does not yet listen to the network events, however, it has hooks that can be extended to support the switching between active connections.

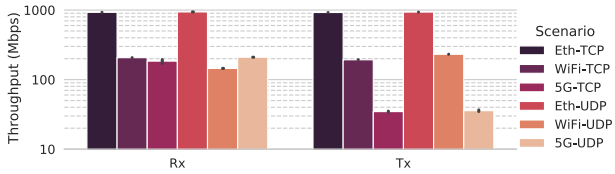


FIGURE 4. iPerf. *Baseline link capabilities for Rx and Tx.*

For each test we report the values measured at our client laptop. Furthermore, we measure the values observed when the client is transmitting the data to the server (Tx), and when it is receiving the data from the server (Rx). Notable here is the cellular connection, whose bandwidth depends on if the laptop is sending or receiving because sending data over the cellular connection we used is slower than receiving data over the same connection.

D. RESULTS

1) BASELINE PERFORMANCE

We present the results of our baseline measurements in Figure 4 and Figure 5.

In Figure 4, we observe that iPerf can reach roughly 940 Mbps over Ethernet in both directions, and 200 Mbps over WiFi. Over 5G links, iPerf can reach 190 Mbps download and 35 Mbps upload in our testbed. The asymmetry between download and upload is due to cellular network technologies. These results provide us with the maximum achievable throughput without overheads caused by MULTI or other protocols in our testbed.

In Figure 5(a) we observe that the time to establish a connection is smaller when using Ethernet compared to WiFi, and the most time by a large margin is when using 5G. We believe that this is because of the larger RTTs for WiFi and 5G, and also due to radio characteristics of the wireless links that can cause the first packet to incur a larger latency than the subsequent packets [34]. Furthermore, when using the base asyncio library we observe that the time to open a TCP connection (2.09 ms over Ethernet, 5.57 ms over WiFi,) is smaller than that for a QUIC connection (31.53 ms over Ethernet, 34.32 ms over WiFi, and 50.2 ms over 5G) because QUIC also performs a TLS handshake during connection establishment.

In Figure 5(b), we observe the MULTI and the baseline TCP and QUIC libraries require similar amount of time to transmit 25 kB; the only notable difference is when MULTI receives data over QUIC. This is most likely a consequence of the fragmentation caused by the MULTI segment header. The additional bytes required by the MULTI header increases the number of IP packets, which in turn increases the number of asynchronous events and calls to the SSL libraries in the QUIC implementation. This increase in events coupled with the slower laptop CPU results in the additional time.

In Figure 5(c), we observe that using the asyncio libraries (Base) results in a significant decrease in the throughput compared to iPerf, and this decrease is significantly large for QUIC; for instance, we were able to achieve only 844 Mbps Tx (10.2% decrease) and 724 Mbps Rx (22.9% decrease) when using TCP over Ethernet, and 48 Mbps Tx (94.8% decrease) and 78 Mbps Rx (91.7% decrease) when using QUIC over Ethernet. We also observe a similar decrease in throughput for WiFi and 5G. We believe that the difference in the Tx and Rx throughput is because of slower laptop CPU coupled with the increased CPU load incurred when reading data at the client; a write results in writing to an asyncio stream buffer, while a read results in asyncio events for reading the requested amount of data from the socket buffers. We also observe that MULTI incurred a small overhead (less than 5% decrease in throughput) when using TCP for Ethernet and WiFi when compared to the base line asyncio, but it incurred a high overhead (up to 20% decrease in throughput) when using QUIC over Ethernet and WiFi. When using 5G, we observe that the performance of MULTI degrades considerably when receiving data over TCP. This is primarily due to the increased number of asyncio events created by MULTI to process the incoming TCP segments and longer latency over the 5G link (46 ms compared to less than 2 ms over WiFi or Ethernet). For WiFi and Ethernet, the network latency is small enough for the asyncio event loop to batch the events together, however this batching does not happen because of the delays and the variance in the delays in the 5G network we used. We are currently exploring approaches to address this shortcoming.

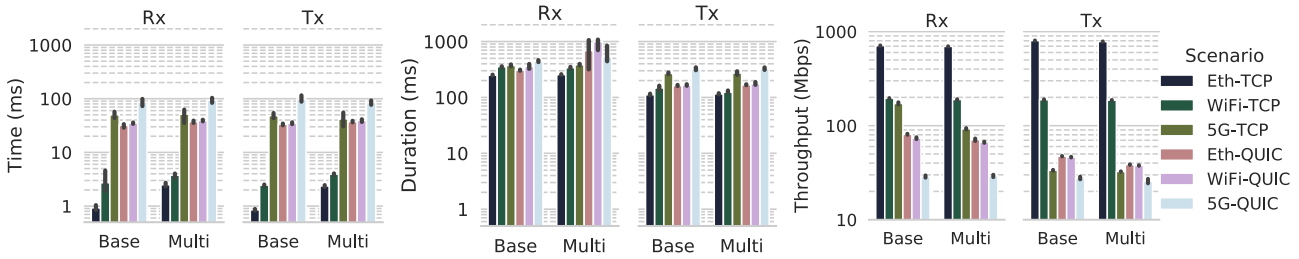
2) MULTI: THREE CONNECTIONS USED TO DEMONSTRATE ITS MULTIPLEXING CAPABILITIES

In this test, we measure the performance of MULTI using three different link-layer transports, namely Ethernet, WiFi, and 5G. We use these tests to demonstrate the multiplexing capabilities of MULTI and present its performance when using multiple available link-layer and transport-layer protocols. We expect to see poor performance because of the inherent weakness of the basic Round Robin scheduler with highly asymmetric links.

In Figure 6(a) we observe that the time to open three connections (for combinations of TCP and QUIC) is longer than the time observed to open a single connection in Figure 5(a). The time to open three connections is the smallest when all the connections use TCP because QUIC requires the TLS handshakes; we plan to investigate the impact of using TLS over TCP in our future work.

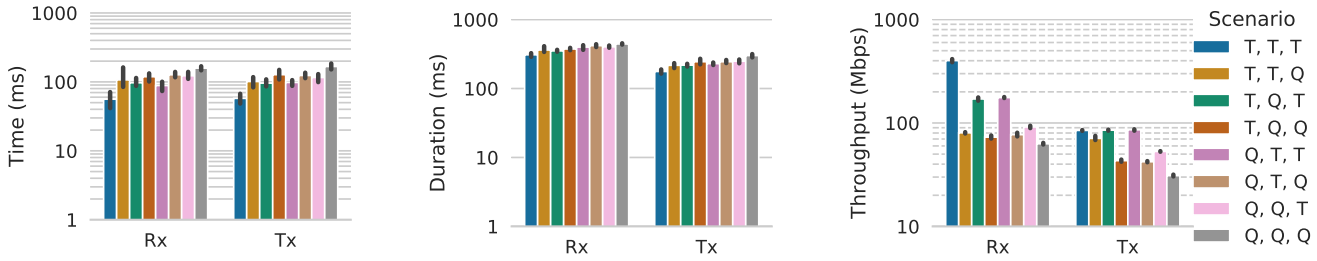
In Figure 6(b) we observe that using multiple links increases the duration to transfer 25 kB of data. This is inline with other works that have compared the effect of multipath protocols for small and large file transfers [12], [33]. Furthermore, sending the data is faster than receiving because of the slower laptop CPU coupled with the WiFi and 5G savings.

Figure 6(c) highlights the benefits and shortcomings of a Round Robin scheduler: it can alleviate the load on the slower links, but the achieved throughput is significantly smaller compared to when using only the link with the best performance. Note that our testbed is biased towards the 1 Gbps



(a) Connection/Session establishment. (b) Duration to transfer 25 kB of data. (c) Throughput when transferring 256 MB of data.

FIGURE 5. Baseline results. *MULTI results represent its bandwidth overheads when using only a single transport layer connection. Figures (a), (b) and (c) respectively present the time to open a connection, the duration to transfer 25 kB of data, and the throughput to transfer 256 MB data when using the base asyncio library (left), and MULTI (right). Error bars represent the standard deviation across 10 iterations.*



(a) Time to open a MULTI session. (b) Duration to transfer 25 kB. (c) Throughput when transferring 256 MB.

FIGURE 6. MULTI performance with three connections. *These results represent MULTI's multiplexing capabilities when using the Round Robin scheduler after waiting for all transport connections to be established. Figures (a), (b) and (c) respectively present the time to open a connection, the duration to transfer 25 kB of data, and the throughput to transfer 256 MB data. MULTI's Round Robin scheduler distributes the load across the connections. Error bars represent the standard deviation across 10 iterations. The color denotes the transport protocol (TCP = T, QUIC = Q) used by Ethernet, WiFi, and 5G.*

Ethernet. For instance, the performance for *T, T, T*, i.e. all TCP, is best as expected, although due to the nature of Round Robin scheduler the speed is not as good as with a single TCP over Ethernet connection. However, the throughput is roughly twice as fast as the TCP over WiFi or 5G, showing the benefit of aggregation. MULTI's scheduler can be tuned to account for the costs of using each link, and the desired performance. For instance, if the faster link is expensive, MULTI can be tuned to minimize the cost while achieving the desired throughput.

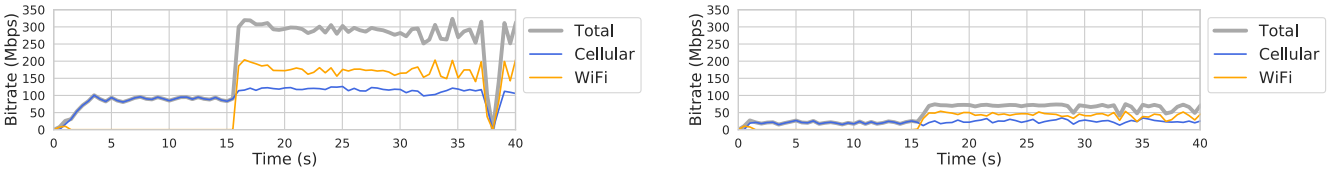
3) MOVING BETWEEN NETWORKS

In Figure 7(a) and Figure 7(b), we present the results of MULTI aggregating bandwidth with a 5G and WiFi network when downloading data from the Internet. In these tests, MULTI first uses cellular network to establish a connection and start transferring data. At 15 seconds, the WiFi network becomes available, and MULTI aggregates both interfaces for additional bandwidth. In the optimal case, MULTI should reach similar speeds as in Figure 5(c), i.e. if we aggregate TCP over WiFi and cellular, the best download speed MULTI should reach is around 350 to 400 Mbps. However, we do not achieve these speeds because our prototype is currently single-threaded, and there are overheads in handling multiple streams simultaneously in single threaded applications.

In Figure 7(a) we show MULTI transferring data over TCP. In the beginning, MULTI reaches a bandwidth of around 100 Mbps over 5G. At the 15 seconds mark, MULTI aggregates the WiFi network to the existing connection. After aggregation, MULTI gains additional bandwidth and reaches the maximum bandwidth of 340 Mbps. This bandwidth decreases to roughly 300 Mbps after a short time, most likely due to full receiving buffers at the client laptop. We also observe some abnormal throughput in WiFi connection. We believe that this could be due to various factors including a) buffers filling in our AP or the test laptop, or b) background traffic in our networks, and c) background activities in our laptop.

In Figure 7(b) we show results when MULTI only uses QUIC to transfer data. At the 15 seconds mark, we add another QUIC connection and start aggregating the connections. Here MULTI reaches its peak performance of 75 Mbps. We also observe a small drop in 5G bandwidth during the experiment. We believe this caused by the interaction of two independent QUIC connections, higher latency of tens of ms over 5G, and the asyncio event loop of the Python.

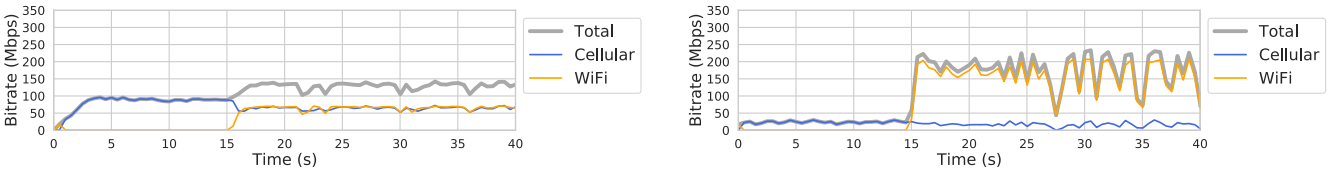
In Figure 8, we show the performance of MULTI when we use a mix of TCP and QUIC connections over 5G and WiFi. In Figure 8(a), we use TCP over 5G and QUIC over WiFi. When we aggregate QUIC over WiFi to TCP over 5G, we observe a drop in the throughput over our 5G link. While QUIC over WiFi reaches similar throughput as in Figure 7(b), the throughput of TCP over 5G drops to 50 Mbps. As above, we believe this drop to be due to the client handling the QUIC stream and the difference in bitrate and latency between the 5G and the WiFi connections. These lead to



(a) Two aggregated transport links over TCP.

(b) Aggregating, both QUIC.

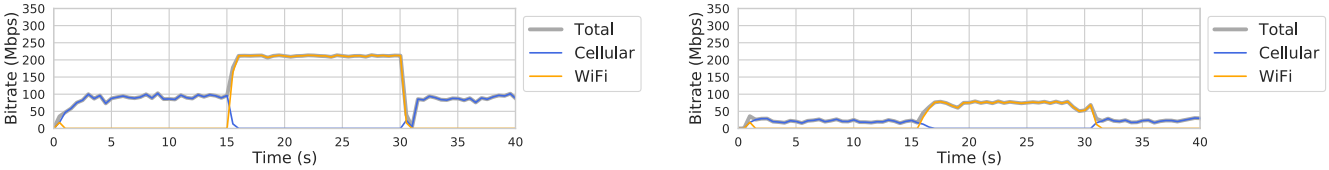
FIGURE 7. MULTI performance when aggregating connections with TCP. *Figures (a) and (b) respectively present how MULTI behaves when moving from 5G network to the range of a WiFi network, either using TCP or QUIC for both connections. In Figure 7(a), after MULTI aggregates 5G and WiFi, it almost doubles available bandwidth. In Figure 7(b), MULTI gains additional bandwidth after aggregation with QUIC. However, QUIC's congestion window increase is slower over 5G due to longer latency. Similarly, QUIC over 5G loses bandwidth possibly due to heterogeneous links and MULTI implementation.*



(a) Two aggregated transport links, 5G-TCP, WiFi-QUIC.

(b) Aggregating 5G-QUIC, WiFi-TCP.

FIGURE 8. MULTI performance when aggregating connections using TCP and QUIC simultaneously. *Figures (a) and (b) respectively show how MULTI behaves when using either TCP over WiFi and QUIC over 5G or vice versa. In Figure 8(a), after MULTI aggregates TCP over 5G and QUIC over WiFi, while MULTI increases overall bandwidth, we observe a drop in TCP over 5G, most likely due to queuing at client. In Figure 8(b), MULTI gains additional bandwidth after aggregation. Variance in WiFi is most likely due to buffering at AP or at the client.*



(a) Switching between links, TCP.

(b) Switching between links, QUIC.

FIGURE 9. MULTI behaviour when switching between networks. *Figures (a) and (b) present how MULTI behaves when switching between 5G and WiFi network. A switch is triggered every 15 seconds. In Figure 9(a), we show the behavior when both connections use TCP. In Figure 9(b), we show behavior with QUIC. Here, we see how QUIC's congestion window increase over 5G takes longer than over WiFi.*

the client receiving more MULTI segments encapsulated in QUIC frames over WiFi, which take longer to process than plain MULTI segments over TCP.

Similarly, Figure 8(b) shows the case where we initiate the connection with QUIC over 5G and aggregate it with TCP over WiFi. As earlier, QUIC over 5G rises to a bandwidth of around 30 Mbps. However, when the TCP over WiFi is added, we again observe the drop in QUIC over 5G due to the above-mentioned reasons. The WiFi throughput of the WiFi connection also becomes unstable at around 23 seconds, most likely due to the behaviors previously mentioned.

In Figure 9, we show how MULTI behaves when we manually switch between transport links. Here we use make-before-break when switching the transports, i.e. the connection over the other link is established before the other link is broken.

In Figure 9(a), we show how MULTI behaves when we switch TCP over 5G to WiFi and vice versa. MULTI achieves the same performance as before when single interface is used. Notable here is the TCP windows size adjustment. In the beginning, the TCP window size slowly increases to reach the maximum bandwidth over 5G. However, when we perform the switch back to 5G from WiFi, the window size converges much more rapidly.

Figure 9(b) presents the results of a similar experiment with both transports using QUIC. Here MULTI reaches the speeds as above using single transports. When the switch between networks happens, we see overlap between connections due to buffers not being empty. As we use make-before-break in our testbed, this allows MULTI to empty buffers instead of losing data.

The results in Figure 7, Figure 8, and Figure 9 demonstrate how MULTI can be used when devices switch between networks. A caveat of our evaluation is that a make before break used in our evaluation may not be possible without the help of the network. However, when moving into the WiFi range and switching to it from 5G, make-before-break should always be possible as long as the 5G (or slower) coverage is available. When moving from WiFi to 5G, the WiFi connection loss can be more abrupt, however, if MULTI uses hooks to the

underlying network interfaces, MULTI could detect impending connection loss from WiFi signal strength and trigger the switch automatically.

V. DISCUSSION AND FUTURE WORK

Our evaluation shows that MULTI can offer programmable multiconnectivity over multiple interfaces and transports. Although our current prototype highlights the strengths of MULTI's architecture, it also shows the weaknesses of the prototype. There are many avenues to optimize its performance; for instance, we were unable to use uvloop [35] because it currently does not support binding UDP sockets to specific interfaces.

Currently our solution's network view is local to the devices on which MULTI is running. We envision a larger system, where different network controllers are available with which MULTI can exchange information to assess the available capacities of the networks. By working together with these controllers, and adding new capabilities to MULTI, we envision that programmable multiconnectivity can be achieved.

A. COUPLED CONGESTION CONTROL

Transport protocols such as TCP and QUIC implement their own congestion control mechanisms. These mechanisms work on single connection basis, i.e. each connection has its own view of congestion. However, when multipath transport protocols such as MPTCP are used, there is a need for coupled congestion control mechanisms as each path will have different congestion characteristics [36].

The current version of MULTI does not support coupled congestion control. However, coupled congestion control can be implemented in user space, as demonstrated by Multipath QUIC (MPQUIC). Specifically, MPQUIC uses Opportunistic Linked Increase Algorithm (OLIA), which we are planning to implement to allow MULTI to handle congestion better [37]. However, some of the challenges we envision include the impact of having a congestion control algorithm in user space when using transport protocols such as TCP whose congestion control algorithms are implemented in the OS kernel.

B. NETWORKING APIs

MULTI exposes two handles: one for reading data that arrives from the network, and the other for sending data to the remote peer. Internally MULTI uses the socket file descriptions via the asyncio python library. MULTI's API are inspired by URLSession [31] and the APIs recommended by the IETF TAPs architecture [25]. Specifically, MULTI's API complements these efforts that are aimed at hiding the semantics of the socket API from the application developers. We acknowledge that the current APIs in our prototype are not comprehensive because they were primarily designed to implement our prototype of MULTI.

C. END-TO-END SUPPORT

Our current MULTI prototype requires both endpoints to support MULTI. Currently, the application developer has to have the knowledge that the destination is also using MULTI. In the future, we envision MULTI to be able to determine if the destination is either MULTI-capable or not through methods similar to TAPS, MPTCP, and Happy Eyeballs. This will allow MULTI to be properly agnostic to the underlying networks.

Similarly, the work on pluginizing QUIC is relevant to MULTI [38]. Pluginized QUIC allows endpoints to exchange protocol plugins per connection basis, thus extending features that either endpoint supports. Protocol plugins would allow MULTI to use new userspace protocols that are not available when the application is created.

D. MIDDLEBOXES

Different multiconnectivity protocols have issues with middleboxes because many middlebox implementations do not behave well with protocols they do not support [9], [12], [38]. As such, a single multiconnectivity protocol may not be able to use all available networks. For example, if a device is connected to two networks, one of which supports MPTCP and one that does not understand MPTCP TCP options, it discards the MPTCP packets. In this instance, MPTCP is restricted to only single path, while MULTI can use both networks as MULTI can use more than one protocol simultaneously. At the same time encrypted communications makes traversing middleboxes even more difficult, as we discuss below.

E. ENCRYPTED COMMUNICATIONS

Encrypted communications have become the norm. For example, QUIC has been designed to use encryption from its inception. QUIC encrypts most of the packet payload while leaving enough of the headers unencrypted for middleboxes to detect QUIC protocol [10].

The goal of the MULTI is similar, encrypt the data of the flows when requested. However, this causes MULTI to encrypt data twice when using underlying protocols that encrypt data like QUIC. This is problematic when dealing with load balancers, as the balancers see QUIC packets which contain encrypted MULTI frames. This can cause different flows of MULTI to reach different servers. This requires MULTI to decide which connection to use based on requested connection priorities and marking the connection to the other server as invalid. While not fatal for MULTI, this is not optimal and requires further study.

F. HEAD-OF-LINE BLOCKING

Head-of-Line blocking is an issue when dealing with network connections that stream buffered data over heterogeneous networks. In MULTI, we can employ similar methods like MPQUIC to avoid the Head-of-Line blocking [12]. MPQUIC uses the same packet scheduler as MPTCP uses in the Linux kernel with few modifications. These include different methods such as specific WINDOW_UPDATE frames to estimate the latency and the bandwidth of different paths. It then uses the results of these estimates to adjust its packet scheduler to avoid Head-of-Line blocking. However, this approach will

not work when using TCP as the transport for exchanging streams, and MULTI can be configured to avoid using TCP when used by applications that are sensitive to Head-of-Line blocking.

G. COSTS OF SCHEDULING AND PAYLOAD INCREASES

Using multiple transports simultaneously carries its own costs. As with any multipath protocols, scheduling traffic over multiple paths inherently incurs additional latency. [12] This latency can be mitigated with scheduler designs, where more complex schedulers can take the differences of different transport into account and optimize the scheduling decisions.

Similarly, using a session layer protocol increases the packet sizes as the headers need to carry enough information for endpoints to associate packets with the connections they belong to. This is true for other multipath protocols regardless of which network layer they operate. For example, the MPTCP carries Connection IDs and other information as TCP options [13]. Like with the scheduler design, the headers can be designed to only carry as much information as is required for the protocol to operate. To achieve this, QUIC uses different sets of headers for connection establishment and data transfer [10].

H. CAVEATS OF THE MULTI

Our MULTI prototype that we used in our evaluation is not perfect. Instead, its main purpose is to highlight both the problems of the current state of multiconnectivity and the insights that can be gleaned from the proof-of-concept implementation. As discussed in earlier chapters, none of the existing multiconnectivity protocols and solutions work in all scenarios. Some of them come close, for example QUIC is already in use and its popularity is ever increasing, and MPTCP has been deployed on some mobile phones. TAPS and NEAT also show how systems can move from being restricted to a single protocol to protocol agnostic system.

MULTI on the other hand tries to combine multiple protocols into a unified connection over multiple interfaces. It is not restricted by the protocols themselves, but is restricted in other ways as discussed in this section. There are several large issues with MULTI; namely congestion control and HOL blocking.

Our MULTI prototype is based on asyncio which has its own performance issues. Optimizations such as uvloop address some of these issues, but they are not a complete replacement for the native asyncio python libraries. For instance, we could not use uvloop and bind UDP sockets. Our prototype is therefore a proof of concept and is not designed to optimize the I/O throughput.

VI. CONCLUDING REMARKS

Multiconnectivity, be it either multihoming or multipath, has been studied for decades [18]. In this paper, we discuss solutions that operate on different layers of the network stack and highlight some key issues applications may face when using them. For example, MPTCP is a good replacement for TCP,

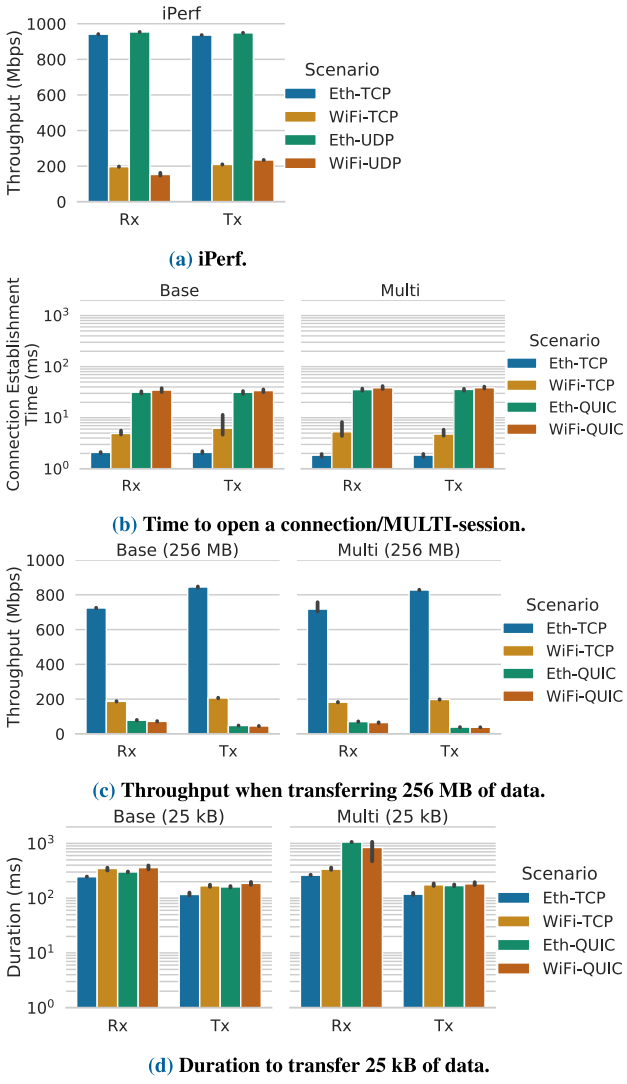
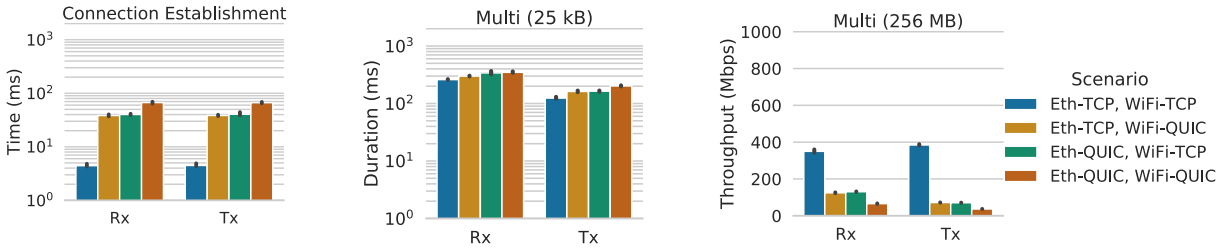


FIGURE 10. **Baseline results (MULTI results are representative for best case).** ***Figure (a) presents the throughput when using iPerf. Figures (b), (c) and (d) respectively present the time to open a connection, the throughput to transfer 256 MB data, and the duration to transfer 25 kB of data when using the base asyncio library (left), and MULTI (right). Error bars represent the standard deviation across 10 iterations.***

but its deployment is hindered by the Internet ossification and middleboxes.

Existing solutions do not offer applications and users the control over the set of chosen communication interfaces. Insights from the seminal work of Bahl et al. [39] and the recent works on TAPS motivate us to believe that this level of control will be vital and useful in the next generation networks and end-user applications.

We therefore created our solution, MULTI, that allows applications to specify their requirements and can be extended to request the network to fulfill them. It is agnostic to the underlying network and transport protocols. It draws from QUIC and MOSH on how to handle the connectivity and roaming in the session layer, i.e. closing a link will not break a connection as long as there is an alternative route.



(a) Time to open a MULTI session.

(b) Duration to transfer 25 kB.

(c) Throughput when transferring 256 MB.

FIGURE 11. MULTI performance with two connections (representative for worst-case by using the Round Robin scheduler after waiting for all transport connections to be established). *Figure (a) presents the throughput when transferring 256 MB, while Figure (b) presents the duration to transfer 25 kB of data. In both cases, MULTI used the Round Robin scheduler to distribute the load across the connections. Error bars represent the standard deviation across 10 iterations.*

While MULTI can currently exchange a single data stream inside multiple connections, we plan to extend it to multiplex multiple streams.

We also envision a system where networks can provide information to MULTI. For example, MULTI could use MAMS [40] to negotiate the required QoS/QoE with the networks when selecting the set of communication interfaces and protocols to use.

Although our prototype highlights the strengths of MULTI's architecture, there are many avenues to optimize its performance; for instance, we were unable to use *uvloop* [35] to replace the default Python event loop as it currently does not support binding UDP sockets to specific interfaces. This shows in poor performance when using QUIC.

Although our MULTI prototype is not optimized, it still highlights the gains that can be achieved with programmable multiconnectivity.

APPENDIX A
ARTIFACTS

The MULTI prototype and results are available at: https://version.helsinki.fi/multiconnectivity/multiconnscratch

APPENDIX B
RESULTS WHEN USING IPv4

In this section, we present some of our experimental results for highlighting that MULTI is agnostic to underlying IP protocol. These results use the same test environment as results presented in §IV with a few differences. As such, there are only two available transport interfaces for MULTI to choose from because the experiments were conducted using IPv4. The test environment does not include a 5G connection because of the double NAT issue discussed in §IV.

In Figure 10, we observe that iPerf and MULTI can reach similar speeds and bandwidths with IPv4 as presented in Figure 4 over IPv6.

Figure 11 shows the results of MULTI using two connections over different combinations of interfaces and protocols.

In Figure 11(a) we present the time to open both the connections. The connection establishment time is the smallest when both the connections use TCP, and the worst when both the connections use QUIC because of the TLS handshakes performed by QUIC. As expected the time to open two TCP connections is twice the amounts observed in Figure 10(b); we observe a similar behavior for QUIC.

In Figure 11(b) we present the time to transfer 25 kB, we observe that MULTI performs significantly better when receiving data over two QUIC connections. However, we believe this to be an artifact of our proof-of-concept and not the real case.

In Figure 11(c), we show the benefits and shortcomings of using the round-robin scheduler. While this scheduler can be used to alleviate the load on the WiFi links, the achieved throughput is significantly smaller compared to when using only the link with the best performance (Ethernet link shown for MULTI in Figure 10(c)). For instance, the performance for *Eth-TCP, WiFi-TCP* is best as expected, although due to the nature of Round Robin scheduler the speed is not as good as with a single TCP over Ethernet connection. However, the throughput is roughly twice as fast as the TCP over WiFi, showing the benefit of aggregation.

REFERENCES

[1] *Android Help—Connect to Wi-Fi Networks on Your Android Device*. Accessed: Mar. 26, 2021. [Online]. Available: https://support.google.com/android/answer/9075847

[2] *Capabilities—Overview of Linux Capabilities*. Accessed: Aug. 26, 2021. [Online]. Available: https://man7.org/linux/man-pages/man7/capabilities.7.html

[3] *Optimize Network Data Usage*. Accessed: Apr. 17, 2021. [Online]. Available: https://developer.android.com/training/basics/network-ops/data-saver

[4] *NoRoot Firewall*. Accessed: Apr. 17, 2021. [Online]. Available: https://play.google.com/store/apps/details?id=app.greyshirts.firewall

[5] K. B. Egevang and P. Francis, *The IP Network Address Translator (NAT)*, document RFC 1631, May 1994. [Online]. Available: https://rfc-editor.org/rfc/rfc1631.txt

[6] W. Reese, "Nginx: The high-performance web server and reverse proxy," *Linux J.*, vol. 2008, no. 173, p. 2, Sep. 2008.

[7] J. Kempf and R. Austein, *The Rise of the Middle and the Future of End-to-End: Reflections on the Evolution of the Internet Architecture*, document RFC 3724, Mar. 2004. [Online]. Available: https://rfc-editor.org/rfc/rfc3724.txt

[8] S. Guha and P. Francis, "An end-middle-end approach to connection establishment," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, New York, NY, USA, 2007, pp. 193–204, doi: 10.1145/1282380.1282403.

[9] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda, "Is it still possible to extend TCP?" in *Proc. IMC*, 2011, pp. 181–194.
[10] J. Iyengar and M. Thomson, *Quic: A UDP-Based Multiplexed and Secure Transport*, document RFC 9000, Internet Requests for Comments, May 2021.
[11] A. Langley, A. Riddoch, A. Wilk, and A. Vicente, "The QUIC transport protocol: Design and internet-scale deployment," in *Proc. SIGCOMM*, New York, NY, USA, 2017, pp. 183–196, doi: 10.1145/3098822.3098842.
[12] Q. De Coninck and O. Bonaventure, "Multipath quic: Design and evaluation," in *Proc. ACM CoNEXT*, 2017, pp. 160–166, doi: 10.1145/3143361.3143370.
[13] C. Paasch and O. Bonaventure, "Multipath TCP," *Commun. ACM*, vol. 2, no. 4, pp. 51–57, 2014, doi: 10.1145/2578901.
[14] K. Winstein and H. Balakrishnan, "Mosh: An interactive remote shell for mobile clients," in *Proc. USENIX*, 2012, pp. 177–182.
[15] C. E. Perkins, "Mobile IP," *IEEE Commun. Mag.*, vol. 35, no. 5, pp. 84–99, May 1997.
[16] *Asyncio—Asynchronous I/O*. Accessed: Mar. 31, 2021. [Online]. Available: https://docs.python.org/3/library/asyncio.html
[17] *Aioquic*. Accessed: Mar. 31, 2021. [Online]. Available: https://aioquic.readthedocs.io/en/latest/
[18] M. Li, A. Lukyanenko, Z. Ou, A. Ylä-Jääski, S. Tarkoma, M. Coudron, and S. Secci, "Multipath transmission for the internet: A survey," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 4, pp. 2887–2925, 4th Quart., 2016.
[19] R. Moskowitz, T. Heer, P. Jokela, and T. Henderson, "*Host Identity Protocol Version 2 (HIPV2)*, document RFC 7401, Internet Requests for Comments, Apr. 2015. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7401.txt
[20] S. Hätönen, T. Huque, A. Rao, G. Jourjon, V. Gramoli, and S. Tarkoma, "An SDN perspective on multi-connectivity and seamless flow migration," *IEEE Netw. Lett.*, vol. 2, no. 1, pp. 19–22, Mar. 2019.
[21] P. Nikander, A. Gurtov, and T. R. Henderson, "Host identity protocol (HIP): Connectivity, mobility, multi-homing, security, and privacy over IPv4 and IPv6 networks," *IEEE Commun. Surveys Tuts.*, vol. 12, no. 2, pp. 186–204, 2nd Quart., 2010.
[22] R. Stewart, *Stream Control Transmission Protocol*, document RFC 4960, Internet Requests for Comments, Sep. 2007. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4960.txt
[23] Y. Cui, T. Li, C. Liu, X. Wang, and M. Kühlewind, "Innovating transport with QUIC: Design approaches and research challenges," *IEEE Internet Comput.*, vol. 21, no. 2, pp. 72–76, Mar./Apr. 2017.
[24] V. K. Adhikari, Y. Guo, F. Hao, V. Hilt, Z.-L. Zhang, M. Varvello, and M. Steiner, "Measurement study of netflix, hulu, and a tale of three CDNs," *IEEE/ACM Trans. Netw.*, vol. 23, no. 6, pp. 1984–1997, Dec. 2015.
[25] T. Pauly, B. Trammell, A. Brunstrom, G. Fairhurst, C. Perkins, P. S. Tiesel, and C. A. Wood, *An Architecture for Transport Services*, document Internet-Draft draft-ietf-taps-arch-10, Working Draft, IETF Secretariat, Apr. 2021. [Online]. Available: https://www.ietf.org/archive/id/draft-ietf-taps-arch-10.txt
[26] N. Khademi, D. Ros, M. Welzl, Z. Bozakov, A. Brunstrom, G. Fairhurst, K.-J. Grinnemo, D. Hayes, P. Hurtig, T. Jones, S. Mangiante, M. Tuxen, and F. Weinrank, "NEAT: A platform and protocol-independent internet transport API," *IEEE Commun. Mag.*, vol. 55, no. 6, pp. 46–54, Oct. 2017.
[27] D. Schinazi and T. Pauly, *Happy Eyeballs Version2: Better Connectivity Using Concurrency*, document RFC 8305, Internet Requests for Comments, Dec. 2017.
[28] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 277–288, Nov. 1984.
[29] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 13–24, 2012.
[30] S. Lee, Y. Shin, and J. Hur, "Return of version downgrade attack in the era of TLS 1.3," in *Proc. 16th Int. Conf. Emerg. Netw. EXperiments Technol.*, New York, NY, USA, 2020, pp. 157–168, doi: 10.1145/3386367.3431310.
[31] Apple. *URLSession—Apple Developer Documentation*. Accessed: Aug. 26, 2020. [Online]. Available: https://developer.apple.com/documentation/foundation/urlsession
[32] *Sans I/O*. Accessed: Mar. 31, 2021. [Online]. Available: https://sans-io.readthedocs.io/
[33] P. Hurtig, S. Alfredsson, A. Brunstrom, K. Evensen, K.-J. Grinnemo, A. F. Hansen, and T. Rozensztrauch, "A neat approach to mobile communication," in *Proc. Workshop Mobility Evolving Internet Archit.*, New York, NY, USA, 2017, pp. 7–12, doi: 10.1145/3097620.3097622.
[34] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin, "A first look at traffic on smartphones," in *Proc. 10th Annu. Conf. Internet Meas.*, New York, NY, USA, 2010, pp. 281–287, doi: 10.1145/1879141.1879176.
[35] *Fast Implementation of Asyncio Event Loop on Top of Libuv*. Accessed: Mar. 26, 2021. [Online]. Available: https://pypi.org/project/uvloop/
[36] C. Raiciu, M. Handley, and D. Wischik, *Coupled congestion control for multipath transport protocols*, document RFC 6356, Internet Requests for Comments, Oct. 2011. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6356.txt. http://www.rfc-editor.org/rfc/rfc6356.txt
[37] R. Khalili, N. Gast, M. Popovic, and J.-Y. Le Boudec, "MPTCP is not Pareto-optimal: Performance issues and a possible solution," *IEEE/ACM Trans. Netw.*, vol. 21, no. 5, pp. 1651–1665, Oct. 2013.
[38] Q. De Coninck, F. Michel, M. Piraux, F. Rochet, T. Given-Wilson, A. Legay, O. Pereira, and O. Bonaventure, "Pluginizing quic," in *Proc. ACM Special Interest Group Data Commun.*, New York, NY, USA, 2019, pp. 59–74, doi: 10.1145/3341302.3342078.
[39] P. Bahl, A. Adya, J. Padhye, and A. Wolman, "Reconsidering wireless systems with multiple radios," *ACM Comput. Commun. Rev.*, vol. 34, no. 5, pp. 39–46, Jul. 2004.
[40] S. Kanugovi, F. Baboescu, J. Zhu, J. Mueller, and S. Seo, *Multi-Access Management Services (MAMS)*, document 8743, Internet Requests for Comments, 2020. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8743.txt

SEPPO HÄTÖNEN received the bachelor's and master's degrees in computer science from the University of Helsinki, where he is currently pursuing the Ph.D. degree in multi-connectivity. His research interests include multi-connectivity, programmable networks, and future networks beyond 5G.

ASHWIN RAO received the Ph.D. degree from the DIANA (formerly, Planete) Project Team, Inria Sophia Antipolis, and the master's degree from the School of Information Technology, IIT Delhi. He is currently a Docent (Adjunct Professor) with the University of Helsinki. His research interests include communication networks, distributed systems, privacy, building next generation distributed systems, and performing measurements for getting insights on the dynamics of communication networks with the objective of identifying performance and privacy issues.

SASU TARKOMA (Senior Member, IEEE) is currently a Professor of computer science with the University of Helsinki and the Head of the Department of Computer Science. He has authored four textbooks. He has published over 200 scientific articles. He holds nine granted U.S. patents. His research interests include internet technology, distributed systems, data analytics, and mobile and ubiquitous computing. He is a fellow of the IET and EAI. His research has received several best paper awards and mentions in conferences and publications, such as the IEEE PerCom, the IEEE ICDCS, ACM CCR, and ACM OSR.