"Helping the Internet scale by leveraging path diversity"

Duchêne, Fabien

ABSTRACT

Since its inception in the 60's, the Internet has evolved from a nationwide network interconnecting a handful of nodes, to a worldwide system interconnecting billion of devices. While the network has dramatically expanded in size, it has also grown in term of adoption. In the age of the Internet of Things, everything and everyone is heavily connected to the interconnected network. This growth came with a cost for service providers: the user's expectation in terms of reliability and performances. To achieve reliability and performance, network operators and engineers designed redundant systems and tried to balance the load across the network's different paths. While adding redundant paths into the network was already a challenge, a greater one awaited: efficiently using them. This thesis is a contribution to improve different solutions and leverage the path diversity, especially in datacenters and enterprise networks. First, we improve Multipath TCP to make it compatible with curre...

CITE THIS VERSION

Helping the Internet scale by leveraging path diversity

Fabien Duchêne

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

September 2019

ICTEAM
Louvain School of Engineering
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis Committee:

Pr. Olivier **Bonaventure** (Advisor)	UCLouvain/ICTEAM, Belgium
Pr. Laurent **Mathy**	Université de Liège, Belgium
Pr. Charles **Pecheur** (Chair)	UCLouvain/ICTEAM, Belgium
Pr. Ramin **Sadre**	UCLouvain/ICTEAM, Belgium
Pr. Stefano **Secci**	Conservatoire national des arts et métiers, France

Helping the Internet scale by leveraging path diversity
by Fabien Duchêne

ICTEAM
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

This work was partially supported by the ARC-SDN project funded by Communauté française de Belgique and by grants from Google and Facebook.

Does the walker choose the path,
or the path the walker?

— Garth Nix,
Sabriel

Preamble

In 1934, Paul Otlet wrote :

> *On peut imaginer le télescope électrique, permettant de lire de chez soi des livres exposés dans la salle teleg des grandes bibliothèques, aux pages demandées d'avance. Ce sera le livre téléphoté.*[1]
>
> *Ici, la Table de Travail n'est plus chargée d'aucun livre. À leur place se dresse un écran et à portée un téléphone. Là-bas, au loin, dans un édifice immense, sont tous les livres et tous les renseignements, avec tout l'espace que requiert leur enregistrement et leur manutention, [...] De là, on fait apparaître sur l'écran la page à lire pour connaître la question posée par téléphone avec ou sans fil. Un écran serait double, quadruple ou décuple s'il s'agissait de multiplier les textes et les documents à confronter simultanément ; il y aurait un haut parleur si la vue devrait être aidée par une audition. Une telle hypothèse, un Wells certes l'aimerait. Utopie aujourd'hui parce qu'elle n'existe encore nulle part, mais elle pourrait bien devenir la réalité de demain pourvu que se perfectionnent encore nos méthodes et notre instrumentation*[2]

Less than century later, what Otlet envisioned as the "Radiated Library", has become "*the Internet*". Since it's inception in the 60's, the "Advanced Research Projects Agency Network" (**ARPANET**) has evolved from a nationwide network interconnecting a handful of nodes, to a worldwide system interconnecting billion of devices.

While the network has dramatically grown in size, it has also grown in adoption. In the age of the *Internet of Things*, everything and everyone is heavily connected to the *interconnected network*. From the light bulb to the car, our environment and day-to-day life rely more and more on *the Internet*. This growth came with a cost for service providers: the user's expectation in

[1] Otlet Paul, Traité de documentation : le livre sur le livre, théorie et pratique, Bruxelles, Editions Mundaneum, 1934, 431 p.238

[2] Otlet Paul, Traité de documentation : le livre sur le livre, théorie et pratique, Bruxelles, Editions Mundaneum, 1934, 431 p. 428

terms of reliability and performance. A user would not accept that the light does not instantly turn on when he asks his smart watch simply because *"the Internet is broken"*.

To achieve reliability and performance, network operators and engineers design redundant systems and try to balance the load across different network paths. While adding redundant paths into the network was already a challenge, a greater one awaited: efficiently using these different network paths.

This thesis is a contribution to explore and improve different solutions to leverage the path diversity, especially in datacenters and enterprise networks. The main contributions of this thesis are the following:

- *A different point of view on how Multipath TCP uses subflows*

  To this day, Multipath TCP [FRHB13] has been used to improve bandwidth and resilience by leveraging different network paths available. In Chapter 2, we explore different views on how these paths might be used to improve performance, but more importantly give the application more control over the selection process.

- *Enabling in-network bytestream functions*

  With the advent of Software Defined Networks (SDN) [KRV+15b], Network Function Virtualisation (NFV) [LC15] or Service Function Chaining (SFC) [BJSE16], network operators expect their networks to support flexible services beyond the mere forwarding of packets.

  IPv6 Segment Routing [FPG+18] provides enhanced traffic engineering capabilities and is key to support Service Function Chaining (SFC). With SFC, an end-to-end service is the composition of a series of in-network services. Simple services such as NAT, accounting or stateless firewalls can be implemented on a per-packet basis. However, more interesting services like transparent proxies, transparent compression or encryption, transcoding, etc. require functions that operate on the bytestream.

  In Chapter 3, we extend the IPv6 implementation of Segment Routing in the Linux kernel to enable network functions that operate on the bytestream and not on a per-packet basis. Our solution enable network architects to design end-to-end services as a series of in-network functions.

- *Steering transport protocols in Multipath networks*

Because of the way they were designed, current transport protocols struggle to efficiently use the different available paths between a pair of hosts. While solutions like Multipath TCP tried to tackle the problem at the transport layer, TCP and UDP still depend on how routers forward their packet through Equal Cost Multipath (ECMP).

In the first part of Chapter 4, we present FlowBender, a load balancing mechanism that uses ECN and ECMP to dynamically reroute congested flows in Datacenters. The author of this thesis had the opportunity to contribute to this design while interning at Google and the lessons learned from this design deeply affected the remainder of its thesis. In the second part of Chapter 4, we use the lessons learned in the previous chapters to propose a new architecture combining Segment Routing and eBPF to allow transport protocols to benefit from the path diversity.

- *Allowing Multipath TCP to be used in datacenters by making it compatible with current load-balancers and anycast*

One of the major drawbacks of Multipath TCP is that it is not currently compatible with stateless load balancers which rely on the five-tuple for their forwarding decision. This problem has been hindering the deployment of Multipath TCP on servers since the beginning.

In Chapter 5, we describe the problem, and show that this limitation can be circumvented with a small change to the handling of the initial connection. Clients use this connection to discover the load-balanced server and the additional Multipath TCP connections are terminated at a unique address associated to each physical server. With this small change, Multipath TCP becomes compatible with existing stateless load balancers. Furthermore, we show that the same approach enables anycast Multipath TCP services, a major benefit given the difficulty of deploying anycast TCP services.

Bibliographic notes

Conference publications

1. *How hard can it be? Designing and implementing a deployable multipath TCP. USENIX Symposium on Networked Systems Design and Implementation*
   C. Raiciu, C. Paasch, S. Barré, A. Ford, M. Honda, F. Duchêne and O. Bonaventure. USENIX Symposium on Networked Systems Design and Implementation, 2012.

2. *Exploring mobile/WiFi handover with multipath TCP.*
   C. Paasch, G. Detal, F. Duchêne, C. Raicu and O. Bonaventure. CellNet, 2012.

3. *Are TCP Extensions Middlebox-proof?*
   B. Hesmans, F. Duchêne, C. Paasch, G. Detal and O. Bonaventure. ACM CoNEXT - HotMiddlebox, 2013.

4. *Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks.*
   A. Kabbani, B. Vamanan, H. Hasan and F. Duchêne. ACM CoNEXT, 2014.

5. *Making multipath TCP friendlier to load balancers and anycast.*
   F. Duchêne and O. Bonaventure. IEEE 25th International Conference on Network Protocols (ICNP), 2017.

6. *SRv6Pipes: enabling in-network bytestream functions.*
   F. Duchêne, D. Lebrun and O. Bonaventure. IFIP Networking, 2018.

7. *Leveraging eBPF for programmable network functions with IPv6 segment routing.*
   M. Xhonneux, F. Duchêne and O. Bonaventure. ACM CoNEXT 2018,.

Posters and demos

1. *Exploring various use cases for IPv6 Segment Routing*
   F. Duchêne, M. Jadin and O. Bonaventure. ACM SIGCOMM 2018 Conference on Posters and Demos, 2018.

Journal publications

1. *SRv6Pipes: enabling in-network bytestream functions (extended version).*
   F. Duchêne, D. Lebrun and O. Bonaventure. Computer Communications (COMCOM), 2019.

IETF contributions

1. *Multipath TCP Address Advertisement*
   F. Duchene, O. Bonaventure .
   IETF Internet-Draft draft-duchene-mptcp-add-addr-00, 2016
   This draft has been merged in the RFC6824bis referenced below.

2. *Multipath TCP MIB*
   F. Duchene, C. Paasch, O. Bonaventure.
   IETF Internet-Draft draft-duchene-mptcp-mib-00, 2015.

3. *Multipath TCP Load Balancing*
   F. Duchene, V. Olteanu *et al.*.
   IETF Internet-Draft draft-duchene-mptcp-load-balancing-01, 2017
   This draft has been merged in the RFC6824bis referenced below.

4. *RFC6824bis: TCP Extensions for Multipath Operation with Multiple Addresses*
   A. Ford, C. Raiciu *et al.*.
   IETF Internet-Draft draft-ietf-mptcp-rfc6824bis, 2019.

5. *A socket API to control IPv6 Segment Routing*
   F. Duchene, O. Bonaventure.
   IETF Internet-Draft draft-duchene-spring-srv6-socket-00, 2018.

6. *A socket API to control Multipath TCP*
   B. Hesmans, O. Bonaventure, F. Duchene
   IETF Internet-Draft draft-hesmans-mptcp-socket-03, 2018.

Reading IETF draft names

All IETF draft names begin with `draft-`*`name`*`-`*`wg`*`-`. The *`wg`* part is the name of the working group relevant for the draft. When *`name`* is the last name of the draft's main editor, it means that the draft is in an early stage and not yet adopted by its working group. Instead, if *`name`* is equal to `ietf`, then the draft is adopted by its working group and will likely be promoted to RFC status once it reaches a sufficient level of maturity and stability.

Acknowledgments

First, I would like to thank my advisor, Prof. Olivier Bonaventure. By believing in me and giving me the opportunity to join his team, he gave me the possibility to work on a broad range of exciting subjects. His guidance and availability enabled me to move forward and challenge myself throughout the years. I am sincerely grateful for the amount of time we spent discussing to make sure I moved in the right direction. If Olivier's unique perspective allowed me to progress as a researcher, his wisdom helped me improve as a person and for that I am forever grateful.

I would also like to thank my thesis jury, Laurent Mathy, Charles Pecheur, Ramin Sadre, and Stefano Secci, for their insightful comments and the very interesting discussions we had during my private defense.

This thesis is the results of collaborations with other persons. I would like to thank my co-authors, Sebastien Barré, Quentin de Coninck, Gregory Detal, Alan Ford, Benjamin Hesmans, Michio Honda, David Lebrun, Christoph Paasch, Costin Raiciu and Mathieu Xhonneux. I'm especially grateful to Abdul Kabbani, Balajee Vamanan and Jahangir Hasan for giving me the opportunity to work with them on FlowBender. I am also grateful to Ashby Armistead and Nandita Dukkipati for being my managers while I was interning at Google.

I would like to thank my former colleagues of the IP Networking Lab and the INGI department. Among them a special thanks go to David Lebrun for being that sharp minded colleague that always has an exciting idea but more importantly for being a true friend; Christoph Paasch for all the great moments we spent together around the world working on Multipath TCP; Quentin De Coninck for his contribution to my work on Multipath TCP's scheduling and all the discussions we shared; Olivier Tilmans for its numerous insights; Mathieu Jadin for always brightening the mood. I thank also the colleagues I shared an office with (Gregory Detal, Virginie Van den

Schrieck, Sébastien Combéfis, Hoang Tran Viet, François Michel, Maxime Piraux, Pierre Francois, Benjamin Hesmans,...) for all the discussions we had and from which I learned a lot. I would like to thank the staff from INGI: Vanessa Maons, Sophie Renard, Chantal Poncin, Pierre Reinbold, Nicolas Detienne, Anthony Gégo, Ludovic Taffin... for their availability that helped me conduct my research in the best possible conditions.

I want to express my gratitude to my family that supported me and helped me pursue my studies. I also owe a special thank you to my close friends Geoffrey, François and Guillaume for their ~~unbearable never-ending taunting~~ support throughout the years. Many thanks go as well to all the people that have been close to me during these last years. They all contributed to this thesis in their own way.

Finally, I wish to express a heartful thank you to my girlfriend Marie-Marie. She *came my way on a winter day* and gave me the strength to achieve what I sometimes thought was the unachievable. This thesis would never have been possible without her standing by my side.

Fabien

Contents

Chapter 1

Introduction

To cope with the Internet's growth, several solutions have been proposed at different layers of the OSI model. In this section, we give a short description of those that are relevant for this thesis.

1.1 Internet Protocol (IP)

The Internet Protocol (IP) is the principal network protocol in today's Internet. IP is responsible for addressing host interfaces and provides an addressing system. IP is also responsible for the formating of datagrams across the network between a source address and a destination address. Today, two versions of IP are widely deployed: IPv4 and IPv6.

Internet Protocol version 4 (IPv4) [Pos81a] is the fourth version of the Internet Protocol, and the first to have been largely deployed. The IPv4 header is illustrated in Figure 1.1. The relevant fields for this thesis are:

- **Time to Live** this field represents the datagram's lifetime and prevents it from persisting too long inside a network. When a datagram reaches a router, the router decrements the TTL field by 1. When the TTL field reaches 0, the router drops the packet.

- **ECN** These two bits (`ECN` and `ECT`) are used by the Explicit Congestion Nofitication (ECN) [RFB01] that allows end-to-end notification of network congestion without dropping packets.

- **Addresses** are the addresses of the source and the destination of the datagram.

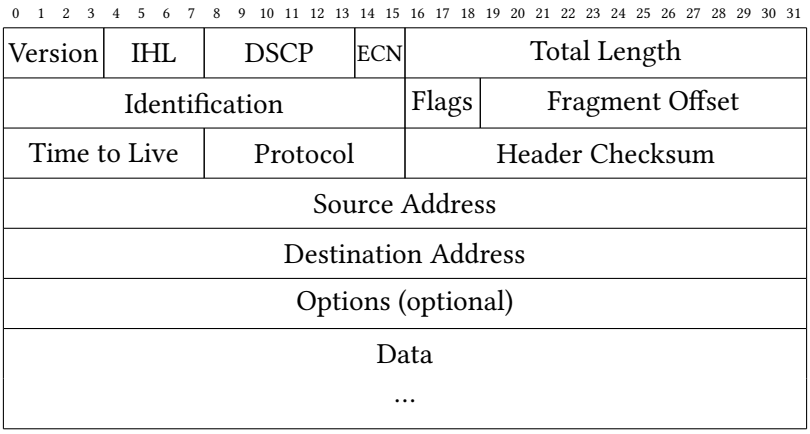


Figure 1.1: Internet Protocol version 4 header (IPv4)

IPv4 address exhaustion As shown in Figure 1.1, IPv4 uses 32-bits addresses. While this was sufficient in the 1980s, the growth of the Internet has since consumed most of the IPv4 addressing space [ICA11]. One of the solution used by network administrators to alleviate the effects of this exhaustion is the deployment of Network Address Translation (NAT) [SE01]. NAT is a method of remapping one IP address space into another by modifying the address in the IP header when a datagram transits between two networks. One of the limitations of this technique is that it breaks the end-to-end principle for the hosts residing behind the NAT. Another solution to the IPv4 address exhaustion is a new version of IP: IPv6.

Internet Protocol version 6 (IPv6) [DH98] is the sixth version of IP. Where IPv4 uses 32-bits addresses, IPv6 uses 128-bits addresses and a simplified header compared to IPv4. This increase in the address length is due to the growth of the Internet and the IPv4 address exhaustion. While its deployment was still pretty slow in 2010 [CGKR10], many ISPs started to deploy IPv6 [Soc11] during the last few years.

1.2 Transmission Control Protocol (TCP)

Developed in the early days of the Internet, the Transmission Control Protocol (TCP) [Pos81b] is still the most used transport protocol nowadays. TCP allows to send an in-order and reliable byte-stream between two hosts. To do so, TCP splits the bytestream sent by the application into segments. These segments are transmitted over the underlying network protocol (IPv4/IPv6) with the TCP header shown in Figure 1.3. To identify both endpoints of the bytestream, TCP uses a 4-tuple: IP addresses (source/destination) and port

numbers (source/destination). To ensure an in-order delivery of the data, TCP assigns a sequence number to each byte of the bytestream. This sequence number is incremented for each byte, allowing the receiver to put the segments back in order upon reception. The reliability is handled by acknowledgments. When a host transmits data, it expects the other end to acknowledge it before a certain amount of time. If the data is not acknowledged after that time, it is retransmitted. This mechanism is known as the Retransmission Timer (RTO). In the TCP header, the acknowledgement number is the number of the next segment expected by the receiver. To initiate a new connection, TCP perform a 3-way handshake. This handshake is illustrated in Figure 1.2.

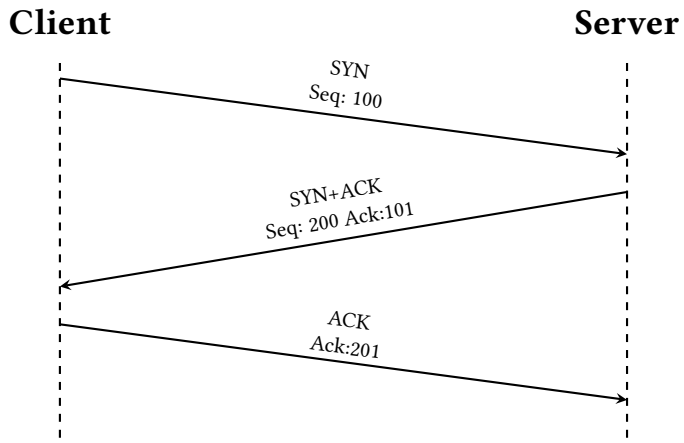


Figure 1.2: TCP 3-way handshake

In this handshake, the client first starts the connection by sending a `SYN` packet with its initial sequence number. If the server accepts the connection, it replies with a `SYN+ACK` packet with its own sequence number and acknowledges the initial sequence number of the client. Then the client sends an `ACK` acknowledging the initial sequence number of the server. After the three-way handshake, the connection is established and both hosts can exchange data in an in-order and reliable way.

1.2.1 Extending TCP: the TCP Options

TCP has been designed to be an extensible protocol. To do so, Figure 1.3 show that TCP has provisioned up to 40 bytes for optional header fields called options.

Options are typically negotiated between hosts during the 3-way handshake. During the 3-way handshake, the client sends a `SYN` with the options it would like to use during the connection. The server answers with a `SYN+ACK` containing the options that will effectively be used during the connection.

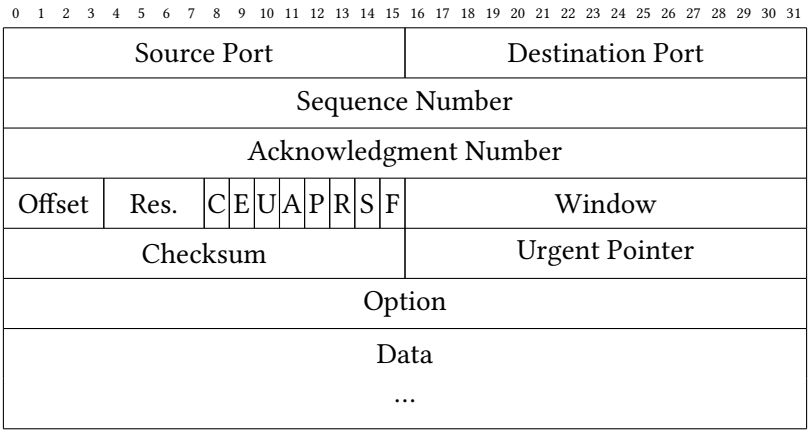


Figure 1.3: Header of the Transmission Control Protocol (TCP)

Options are used to extend the protocol with features like Selective Acknowledgment (SACK) [MMFR96] or Timestamps (MSS) [JBB92].

1.3 Equal Cost Multipath (ECMP)

Equal Cost Multipath (ECMP) is a network routing strategy that allows the traffic between two hosts to be transmitted across multiple paths of equal cost. This is typically used in datacenters using topologies like fat-trees [Lei85] where switches have multiple uplinks but also in enterprise and ISP networks [ACO+06]. When forwarding a packet, the switch/router has to decide which path to use to reach the next hop. A simple solution is a per-packet decision. When a packet reaches the switch, it forwards this packet on one of the possible paths. This can be done using a round-robbin strategy. While this solution is simple to implement, it presents the disadvantage of spreading the packets belonging to a single flow over multiple paths, increasing potential reordering at the receiver's side. To avoid reordering, ECMP is mostly used with a per-flow decision. To ensure that all the packets belonging to a flow use the same path ECMP usually uses a hashing technique that operates on the packet header fields that identify a flow (typically the 5-tuple [cisco]).

1.4 Multipath TCP

To steer packets belonging to different TCP connections through different paths of the network, several solutions have been envisioned at different levels of the ISO/OSI layer. Multipath TCP is a recent TCP extension that tries to solve the problem at the transport layer.

1.4.1 General architecture

Multipath TCP [FRHB13] makes the assumption that to steer a packet along a specific path, the IP address or the port number must change. As this would prevent regular TCP from reconstructing the bytestream, Multipath TCP uses several "regular" TCP connections called *"subflows"* and multiplexes the byte stream over them. The choice of using multiple "regular" TCP subflows and aggregating them comes from the idea that creating a brand new protocol would make it more difficult to deploy because of the ossification of the Internet [HNR+11]. Multipath TCP was designed [RPB+12] to be **deployable without modifying the existing applications or middleboxes**.

This design choice can be seen in Figure 1.4 illustrating Multipath TCP's architecture from an implementation viewpoint. To remain compatible with existing applications, Multipath TCP has been implemented as a layer between the standard socket API and the TCP/IP stack. This way, any application that uses the standard socket API will use Multipath TCP without any modification. When the Multipath TCP layer receives a `send` call from the application, its scheduler will select one of the subflows and use it to send the data. When data comes back from the remote end, the stack will aggregate them and pass them to the application.

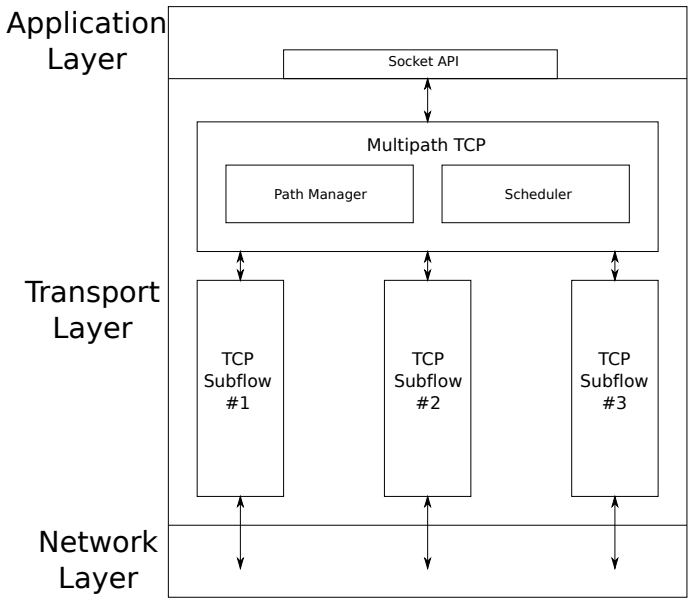


Figure 1.4: Multipath TCP's Architecture

There are two important algorithms in a Multipath TCP implementation:

The Path Manager

The *path manager* shown in Figure 1.4 is the component used to create and terminate subflows. The default path manager, called "full mesh" establishes a subflow per available path. With this path manager, if a client and a server both have two interfaces, 4 subflows will be established.

The Scheduler

The *scheduler* shown in Figure 1.4 is used to elect which subflow is going to be used to send a specific piece of data. When a data is about to be sent, the scheduler selects one of the established subflow based on a specific metric, and transmits the data over that path. The default scheduler uses the subflow with the lowest RTT with an open send window to send a data.

1.4.2 Using the TCP options

During the design of Multipath TCP, a key question concerns how MPTCP metadata should be encoded – embed it in the TCP payload, or use the more traditional TCP options, with potentially problematic interactions with middleboxes [HNR+11]. Within the IETF, opinions were divided, with supporters on both sides [Sch10]. In the end, careful analysis revealed that MPTCP needs explicit connection level acknowledgments for flow control; further, these acknowledgments could cause deadlocks if encoded in the payload [RPB+12]. In reality, there was only one viable choice: using the TCP options.

For Multipath TCP, a new kind of option has been defined and registered with the IANA. The different Multipath TCP options described in this thesis are actually embedded in the TCP options header field. To differentiate between the different types of Multipath TCP options, a subtype is defined.

1.4.3 Establishment of the initial connection

The establishment of the initial Multipath TCP connection is an important part of the protocol. During this stage, the hosts negotiate the utilization of Multipath TCP by advertising the `MP_CAPABLE` option.

To establish the initial connection, Multipath TCP performs a standard three-way handshake. This is shown in Figure 1.6. To determine if the other host supports Multipath TCP, the client sends a `SYN` with the `MP_CAPABLE` option. If the server supports Multipath TCP, it answers with a `SYN+ACK` containing the same option. Otherwise, it simply answers with a `SYN+ACK` and the connection falls-back to regular TCP.

The format of this option is shown in Figure 1.5. In RFC6824 [FRHB13], the `Kind` and `Length` fields are used by TCP to specify that this is a Multipath

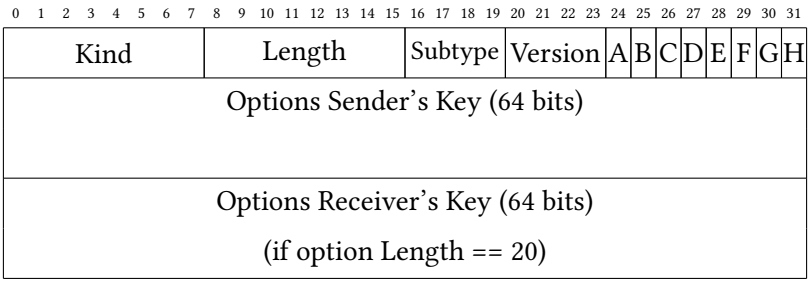


Figure 1.5: Multipath Capable (MP_CAPABLE) Option (RFC6824)

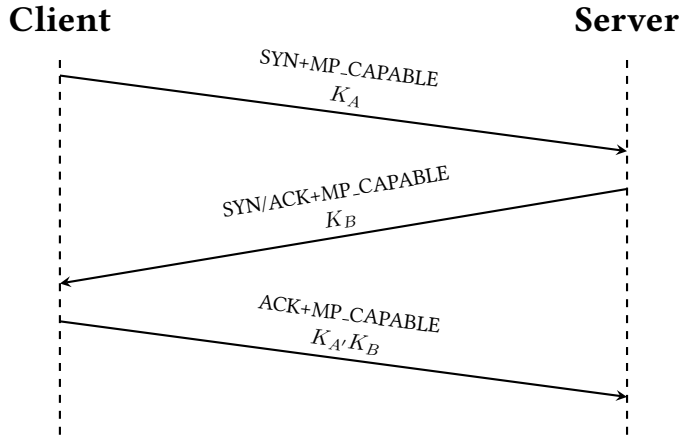


Figure 1.6: Multipath TCP initial connection Establishment

TCP option. The `Subtype` field defines the kind of option, `MP_CAPABLE` in this case. The flags are defined as :

- **A** indicates whether a DSS checksum is required for this connection.
- **B** is an extensibility flag and must be set to 0.
- **C** - **H** are reserved for crypto algorithm negotiation. In this version only the rightmost bit, labeled "**H**", is assigned.

The keys are used as a shared secret to authenticate the hosts during the establishment of additional subflows.

1.4.4 Establishment of an additional subflow

To establish an additional subflow, Multipath TCP also performs a three-way handshake, using the `MP_JOIN` option. This handshake is illustrated in Figure 1.7. To associate this subflow with an already established connection,

Multipath TCP must be able to identify that connection. The intuitive solution would be the use the 5-tuple, but it can not be used as some on-path NAT could modify it. To achieve this goal, Multipath TCP uses a locally unique token, shown as $Token_B$ that has been generated from the key exchanged in the `MP_CAPABLE`. To prevent an attacker from injecting traffic into an already established connection, a HMAC based on a nonce $R_{A/B}$ exchanged with the `MP_JOIN` and the keys exchanged in the initial connection establishment is used.

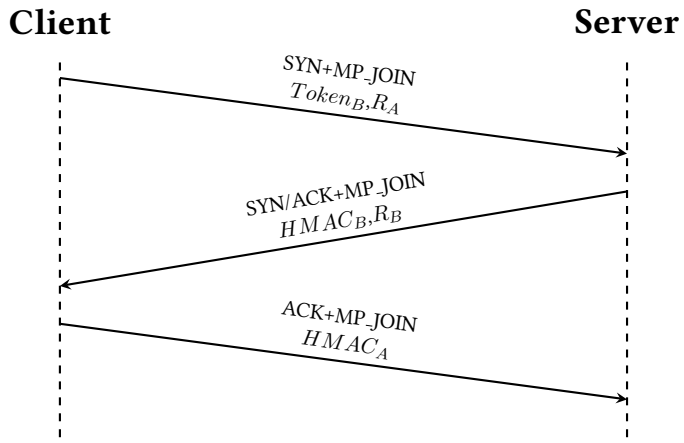


Figure 1.7: Multipath TCP additional subflow establishment

The format of the `MP_JOIN` option used for the `SYN` is shown in Figure 1.8. The fields specific to this option are :

- **B** indicates if the sender wishes this subflow to be used as a backup one. Multipath TCP's backup path system is described in Chapter 2.
- **Address ID** identifies the source address of this packet and has significance within a single connection. It allows address removal and is also used for address management.
- **Receiver's token** is used to identify the connection to join. It is the connection identifier in the remote host.
- **Sender's Random Number** is used for HMAC generation.

1.4.5 Address advertisement

Another critical feature of Multipath TCP is its ability to learn and advertise new addresses. While a client easily knows its own addresses, the addresses of the remote host need to be learned in some way. Figure 1.9 illustrates a

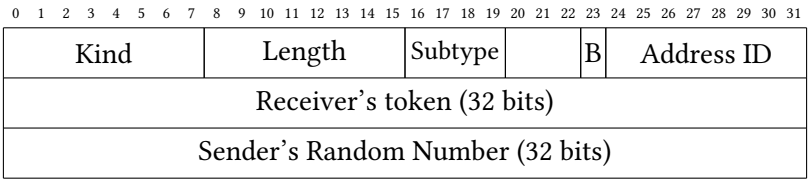


Figure 1.8: Multipath Capable (MP_JOIN) Option for the SYN (RFC6824)

simple scenario in which a smartphone connects to the publicly know IPv4 address of a server 1.1.1.1 (1). The server being multi-homed, it also has another IPv4 address: 2.2.2.2. At this point, the smartphone does not have any knowledge of the server's second interface and thus, cannot establish an additional subflow. The server could use its second interface to initiate a new subflow to the smartphone, but this would most likely fail due to a NAT or a firewall.

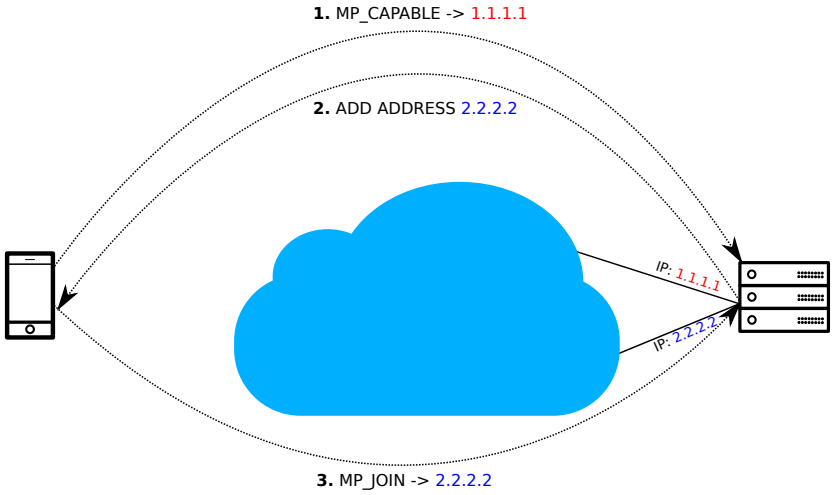


Figure 1.9: Multipath TCP's address advertisement

To circumvent this problem, Multipath TCP supports the possibility for a host to advertise its other IP addresses to the other host. This mechanism is shown in Figure 1.9. As soon as the first connection has been established (1.) the server sends an option to the smartphone (2.) to advertise its secondary IP address 2.2.2.2. At this point, the smartphone knows that the server can be reached via another address. Given that, the smartphone might elect to establish a new subflow to join the connection (3.).

It is important to specify that the address advertisement can work both ways, but as said before, it is likely that if the smartphone had a second interface, it would not be reachable from the server due to a NAT. The advertised

IP address can be an IPv4 address or an IPv6 address, irregardless of the initial subflow's IP version.

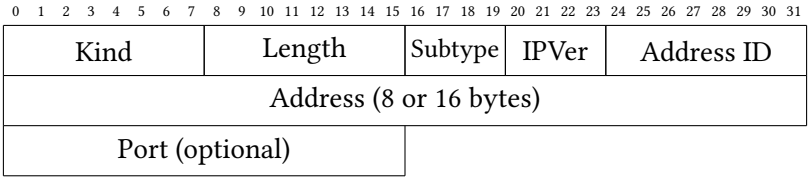


Figure 1.10: Add Address (ADD_ADDR) Option (RFC6824)

The option used to advertise IP addresses is called `ADD_ADDR` and is illustrated in Figure 1.10. In this header, `IPVer` defines the version of IP (4 or 6), `Address ID` represents the internal ID of this address inside to host advertising it. This is useful to remove addresses because this way, when a host removes the address `Address ID`, the receiver just needs to close all connections using this ID. This is necessary because of possible on-path NAT. The `Address` field is the address advertised, and the `Port` is the TCP port to use to establish new connections to this address. It is important to highlight that this `ADD_ADDR` option is not transmitted reliably. If a packet carrying this option is lost, the protocol does not specify any solution to inform the sender, resulting in the definite loss of this information. In that case, the address would end up not being advertised to the other end.

1.5 IPv6 Segment Routing (SRv6)

Segment Routing (SR) is a modern variant of the source routing paradigm. Standardized within the IETF [FPG+18], Segment Routing's main principle is to allow the source of a packet to steer it along an arbitrary path in the network. This path is not necessarily a shortest path. Segment Routing has been designed to be stateless; the path is encoded within the packet in the form of a segment list and a current segment pointer. While Segment Routing has originally been designed to be used on top of the MPLS dataplane [FPG+18], using MPLS labels as segment, the newest flavor uses the IPv6 dataplane to steer packets through an ordered list of *segments* [FDP+19]. There are two types of *segments*: a node segment steers the packet through a particular node in the network while an adjacency segment steers the packet through a particular link in the network. In this thesis we use the word *segment* for both.

Figure 1.11 illustrates a network where multiple paths are available between the **S**ource (S) and the **D**estination (D) of a flow. In this figure, hollow nodes are segment endpoints, as opposed to routers that are not Segment Routing-aware. It is important to note that in this network, **S** and **D** are not

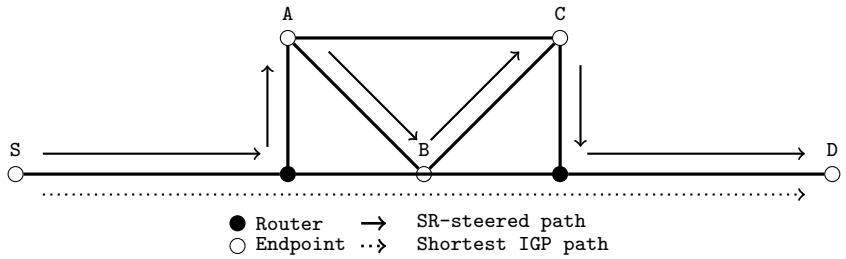


Figure 1.11: Traffic steering between S and D through A, B and C

necessarily the nodes that respectively generated and will receive the packets. They can also be a part of a longest path, where the rest of the path is not Segment Routing-aware. In that case, they act respectively as *ingress* node and *egress* node. In this scenario, a packet egressing `S` will carry a segment list of `A, B, C, D` with the current segment pointer set to `A`. As `A` is the first segment, the packet follows the shortest path to `A`. Upon reaching `A`, the current segment pointer advances to `B` and the packet takes the shortest path to `B`. After going through `C`, the packet reaches `D`. As `D` is the last segment of the segment list it removes the segment list and either consumes the packet (if `D` was the original destination) or forwards it to the original destination. In this configuration, Segment Routing steered the packet over the path `A, B, C, D` where in a non Segment Routing-aware network, the packet would have followed the shortest IGP path.

1.5.1 IPv6 Segment Routing

IPv6 Segment Routing is the IPv6 flavor of Segment Routing. Instead of an MPLS label, IPv6 Segment Routing uses an IPv6 extension header called the Segment Routing Header (SRH). It is illustrated in Figure 1.12.

In this header, the `Next Header` field identifies the protocol following the SRH. The `Header Ext Len` field contains the size of the SRH. The `Routing type` is always set to 4 (it identifies the header as being IPv6 Segment Routing). The `Segments left` field indicates the number of remaining segments. It acts as a pointer in the list and allows to determine the next segment. The `First Segment` contains the index (zero based), in the Segment List, of the last element of the Segment List. The `Flags` are reserved but currently unused. The `Tag` identifies a packet as part of a class or group of packets, e.g., packets sharing the same set of properties. The `Segments List` is a list of segments identified by 128 bits IPv6 addresses in reserve order. The reverse order is explained by Figure 1.13, illustrating the SRH has seen by `S`. In this figure, `A` being the first segment of the path, it is the last address of the list. `D` being the final destination, it is at the first index of the list.

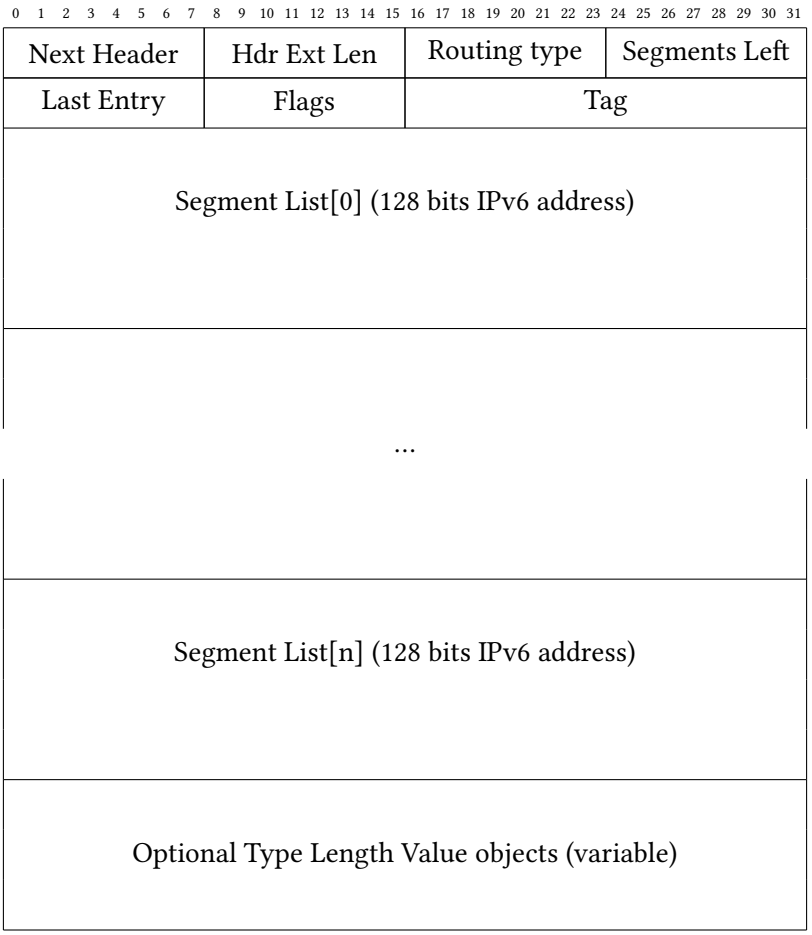


Figure 1.12: IPv6 Segment Routing Header (SRH)

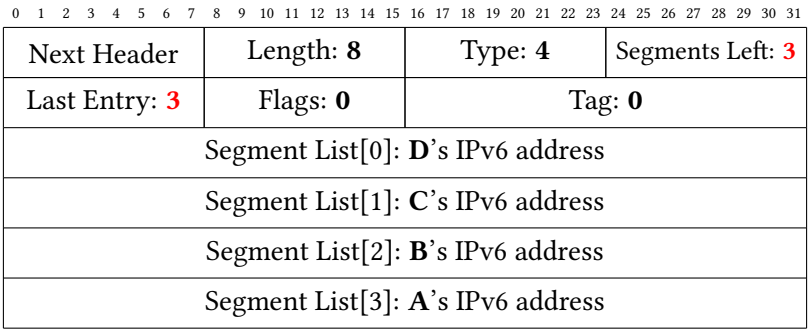


Figure 1.13: SRH representation of 1.11

The Optional Type Length Value objects (TLV) shown in Figure 1.12 are optional headers that can be defined for an SRH. Their goal is to provide meta-data for segment processing. Each TLV has its own length, format and semantic. The only TLVs defined in the specification are HMAC and PAD, used respectively to secure and align the SRH to a multiple of 8 bytes. Additional TLVs can be created to suit specific uses cases, the basic format of a TLV is illustrated in Figure 1.14.

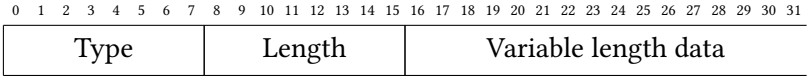


Figure 1.14: Format of a TLV

1.5.2 Processing by segments endpoints

The basic mechanism of IPv6 Segment Routing consists in looking-up in the SRH to determine the next segment and replace the IPv6 destination address by the address of this segment. This is illustrated in Figure 1.15. In this figure, a Client wants to send data to a Server using a path passing through A and B. Thus, the SRH contains (in traversal order) A, B and Server. It is important to mention that it is mandatory to put the IP address of the server as the last segment, otherwise this information would be lost.

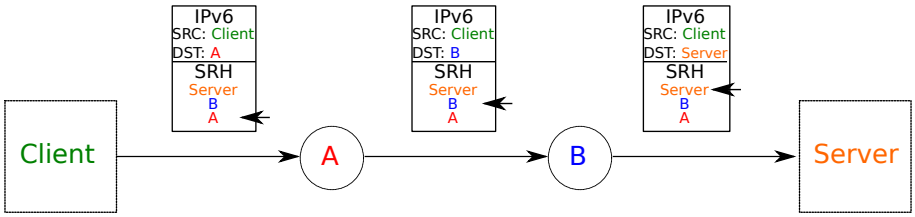


Figure 1.15: IPv6 Segment Routing processing

When a packet leaves the Client, its destination IP address is set to the next segment endpoint A. When reaching A, the endpoint sets the IPv6 destination address to the address of the next node B and decrements the number of Segments Left. This process is repeated until the packet reaches its final destination.

1.5.3 Encapsulation

In the previous sections, we described IPv6 Segment Routing in a scenario where the host generating the packet is IPv6 Segment Routing-aware and directly inserts the SRH *in-line* between the IPv6 layer and the transport layer.

Another solution can be used by an ingress router to impose an SRH onto packets: encapsulation. With encapsulation, the original packet is encapsulated into an outer IPv6 header. In this thesis, we mostly insert the SRH *in-line* because we assume that the origin of the packet supports IPv6 Segment Routing.

Chapter 2

Reconsidering how Multipath TCP handles subflows

2.1 Introduction

In this chapter, we extend Multipath TCP by exploring new ways of using its subflows. In Section 2.2, we reconsider how Multipath TCP handles the backup subflows. In Section 2.3 we propose a trusted resource pooling using Multipath TCP.

2.2 Reconsidering How Multipath TCP Handles Backup Subflows

2.2.1 Different types of subflows

The Multipath TCP specification [FRHB13] supports two types of subflows: *normal subflows* and *backup subflows*. Once a normal subflow has been created, it can be used to send and receive data. A backup subflow is signaled by using either a flag in the `MP_JOIN` option or through the `MP_PRIO` option that can be sent at any time over an active subflow. Smartphones are a typical use case for the backup subflows. Given that cellular usage is often metered, smartphone users often prefer to use the WiFi interface and only use the cellular interface when the WiFi is unavailable. The Multipath TCP specification considers the backup subflows as follows: *path to use only in the event of failure of other working subflows*. Unfortunately, [FRHB13] does not clearly specify how the failure of a working subflow is detected by a host. A simple approach, that is used by the reference Multipath TCP implementation in the Linux kernel [PB$^+$] is to consider that a subflow has failed once the interface used by the subflow fails. This works well on fixed hosts that can react

quickly to the failure of one of their interfaces. However, if the failure lies in the network, the host will wait for n expirations of the TCP retransmission timer before considering the subflow as failed. In wireless networks, there are many situations where a smartphone remains associated with an access points (and thus is assigned to an IP address), but the link quality is so bad that most packets are lost.

In this section, we reconsider how Multipath TCP handles backup subflows. This problem is important for both smartphones and hybrid access networks that are the two ongoing deployments of Multipath TCP [BBG+19]. This section is organised as follows. We first introduce the active backup scheduler in 2.2.2. In 2.2.3, we compare the performance of the active backup scheduler with the default and default backup ones under different types of traffic. We explore related works in Section 2.2.5 and finally discuss our solution in 2.2.6.

2.2.2 The case for packet expirations

The packet scheduler [PFAB14] illustrated in Figure 1.4 is an important component of any Multipath TCP implementation. When several subflows are active, this algorithm decides the subflow that is used to send each data packet. In the reference implementation of Multipath TCP in the Linux kernel [PB+], the default scheduler operates as shown on Figure 2.1. This scheduler prefers the non-backup subflows having an open congestion window and the lowest round-trip-time [BPB11, PFAB14]. It only uses the backup subflows for a specific packet once all the non-backup subflows failed to transmit that packet.

To improve the user experience, we modify this default scheduler by adding an expiration time to each packet. Instead of waiting for a complete failure of all active subflows to use the backup ones as specified in [FRH+19], we agree to use the backup subflows once packets have been delayed for a specified time in the Multipath TCP stack.

Our active backup scheduler offers a trade-off between packet time delivery and the utilization of the backup interface. This trade-off is determined by a new parameter associated to each Multipath TCP connection: the *expiration delay*. To configure the *expiration delay*, we define a new socket option called `MPTCP_EXPIRATION`. As shown by Figure 2.3, the delay can be configured by using this new socket option when creating each Multipath TCP connection inside an application or as a system-wide default.

We implement the active backup scheduler as an extension of the default scheduler. A graphical representation of this scheduler is shown on Figure 2.2. When the application pushes data to the kernel via the `tcp_sendmsg` call, the kernel records the timestamp of the call and associates it to the data. Once the data reaches the scheduler, the scheduling function determines if the data

```
SELECT_SUBFLOW:
list = GET_NON_BACKUP_SUBFLOWS()

/* Get the subflows not already used for
this packet from the list */
list = GET_UNUSED_SUBFLOWS(list, packet)

IF NOT EMPTY list
list = GET_OPEN_CWND_SUBFLOWS(list)

IF EMPTY list
return WAIT_AND_RETRY
ELSE
return BEST_RTT_SUBFLOW(list)

ELSE
list = GET_BACKUP_SUBFLOWS()
list = GET_UNUSED_SUBFLOWS(list, packet)
list = GET_OPEN_CWND_SUBFLOWS(list)

IF EMPTY list
RESET_USED_FLAGS()
return WAIT_AND_RETRY()
ELSE
return BEST_RTT_SUBFLOW(list)
```

Figure 2.1: Default scheduler algorithm to choose a subflow.

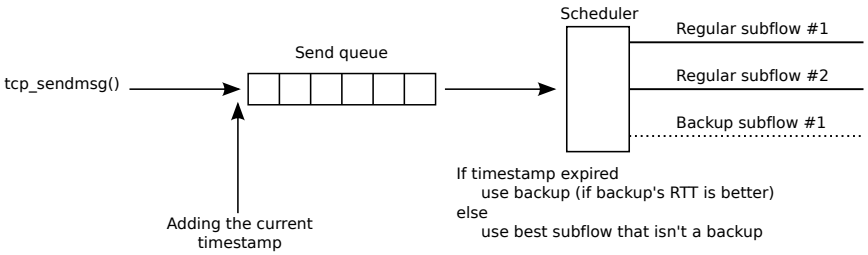


Figure 2.2: Graphical representation of the active backup scheduler.

```
int delay = 40; /* 40ms delay */
setsockopt(sock, SOL_TCP,
MPTCP_EXPIRATION, &delay, sizeof(int));
```

Figure 2.3: Typical way of setting the expiration delay in an application.

spent too much time in the send-queue by computing the difference between the current timestamp and the sum of the recorded timestamp stored with the data and the expiration delay of the connection. While the non-expired data is sent over the non-backup lowest RTT available subflow, the expired data

```
SELECT_SUBFLOW:
IF packet.ts + expiration > now
list = GET_ALL_SUBFLOWS()
list = GET_UNUSED_SUBFLOWS(list, packet)
list = GET_OPEN_CWND_SUBFLOWS(list)

IF NOT EMPTY list
return BEST_RTT_SUBFLOW(list)
ELSE
RESET_USED_FLAGS()
return WAIT_AND_RETRY()
ELSE
/* ... unmodified default code ... */
```

Figure 2.4: Active backup scheduler algorithm to choose a subflow.

is sent over the lowest-RTT available subflow, considering the backups links. This allows that, if some data is considered late by the user parameter, it will always be sent on the fastest available link. If the user does not set any value to `MPTCP_EXPIRATION` or sets a value of 0, the scheduler behaves like the default MPTCP scheduler. To avoid modifying applications, we implemented a `mptcp_sched_expiration` sysctl with a default value of 0. By default, the value of this sysctl will be used upon setting another value via `setsockopt`.

The value of the `MPTCP_EXPIRATION` parameter is local to the host where the expiration has been set. Thus, it controls the packets transmitted by this host. In many situations, the host that sends the data is not the one that wants to influence the utilization of the subflows. A typical example is when a smartphone retrieves data from a server. The smartphone typically wants to consider the cellular interface as a backup one, using it only when the WiFi interface has bad performance or is lost. The server is usually single-homed. For the server, sending data towards the smartphone's cellular or WiFi interfaces has the same cost. To meet the smartphone's user expectations, the server must be able to learn the smartphone's expiration delay.

To achieve this goal, we define a new MPTCP experimental option as specified in [FRH+19] and shown by Figure 2.5. This option carries the value of the expiration delay and is transmitted in a reliable way. Upon reception of the option, the receiver will use the value as the `MPTCP_EXPIRATION` for that connection.

The first 20 bits are used to define a MPTCP option, its length and subtype (experimental). The 'S' and 'U' bits are used to make the option reliable. The "Experiment ID" is the experiment identifier that will be assigned to the `MPTCP_EXPIRATION` option by the IANA. The last 16 bits are used to carry the expiration value in milliseconds.

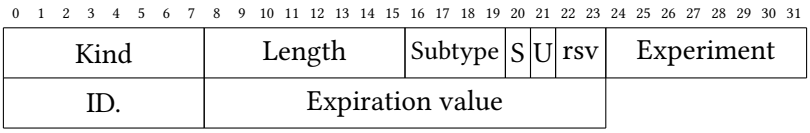


Figure 2.5: Structure of the experimental option

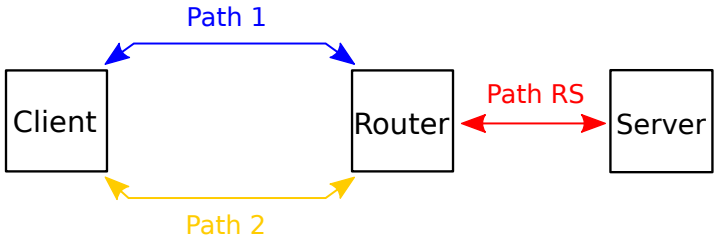


Figure 2.6: Topology used in our tests. The RS link does not introduce any delay/bandwidth constraint.

2.2.3 Evaluation

First, we introduce our methodology used for our measurements. We then explore the performance of the active backup scheduler first in emulated networks without losses, then with different packet loss ratios and finally in random topologies.

2.2.4 Methodology

In the remaining of this section, we analyze the performance of the active backup scheduler in a two-paths topology shown on Figure 2.6 where the bottom one is considered as the backup one. Unless specified, the first path is a 10 Mbps 30 ms RTT link (e.g. a WiFi network) and the second one is a 10 Mbps 20 ms RTT link (e.g. a LTE network). The size of the router buffers is equal to the bandwidth-delay product and they use a FIFO queuing policy.

We use three different types of applications to generate traffic. Our first application is the `wget` HTTP client. We use it to mimic a 20 MB file download. Our second type of traffic is an application rate-limited bulk download using `netcat` [Gia13] and `pipe viewer` [ea15] for a file of 20 MB. Our third application is the loading of a complete web page emulated using `epload` [WBKW14]. We focus here our analysis on the loading of five different web pages, similar results can be observed on other pages given by [WBKW14]. We ensure that their contents are stored in RAM on the server. For each of these applications, we measure the duration of the scenario (data transfer or loading time) and the percentage of bytes that are carried on the

second path in function of the expiration delay. In the `wget` and `epload` cases, we use Apache [FK97] and HTTP1.1 to serve contents on the server. In our experiments, we compare 3 configurations:

- ***Default***: the default Multipath TCP scheduler with two links, none of them configured as backup. In this configuration the scheduler is expected to use both links.
- ***Default with backup***: the default Multipath TCP scheduler with two links with the second path (LTE), configured as a backup path. In this configuration the scheduler is expected to use the backup path only when the first path fails.
- ***Active backup***: our modified Multipath TCP scheduler with two links with the second path (LTE), configured as a backup path. In this configuration, the scheduler is expected to use the backup path when the expiration timer expires.

We run our tests in `mininet` environments [LHM10] under Ubuntu 14.04 on a server with Intel(R) Xeon(R) X5472 @ 3.00GHz and 32 GB of RAM. Each test (except those with random topologies) is repeated at least 5 times.

Non-lossy networks

In this subsection, we observe how the active backup scheduler behaves in networks where there are no random packet losses. This will help us to get a basic understanding of its mechanisms.

Figure 2.7 shows that for a bulk transfer, the active backup scheduler exhibits similar characteristics as the default scheduler without backup link, especially when expiration delay values are low. This is expected since the application pushes data as fast as possible and is thus limited by the network. If the congestion window of the non-backup path is full, the data is backlogged in the MPTCP stack and will sooner or later expire, triggering the utilization of the backup path.

This intuition can be verified by controlling the rate of the bulk transfer. Figure 2.8 shows that the utilization of the backup link grows with the data rate. Since the second path is the lowest RTT one, it is expected to observe most of the traffic on that path when the rate is low, whereas all traffic could be sent on the main path without using the backup one (using here an expiration delay of 35 ms). It is also interesting to see that under a 5 Mbps data rate, the default scheduler balances nearly equally the traffic over both paths while the percentage of bytes on the backup path using the active backup scheduler remains low. When the sending congestion window of one path

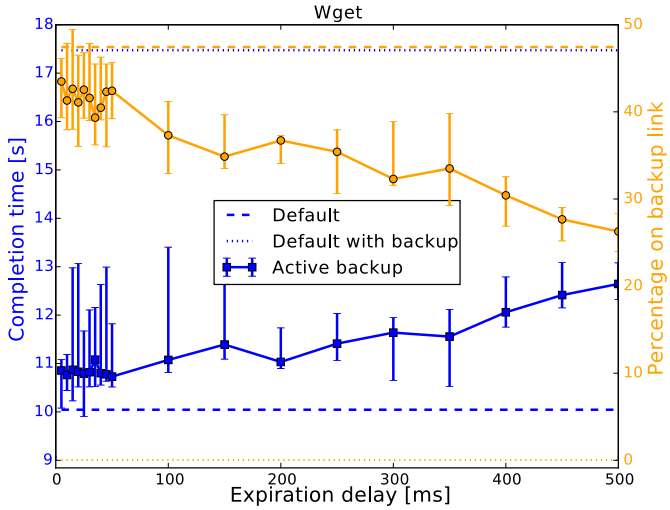


Figure 2.7: Bulk transfer with `wget`. The dots show the median of results for a given expiration delay. The vertical bars shown the minimal and maximal values. Only median values are shown for default and default backup.

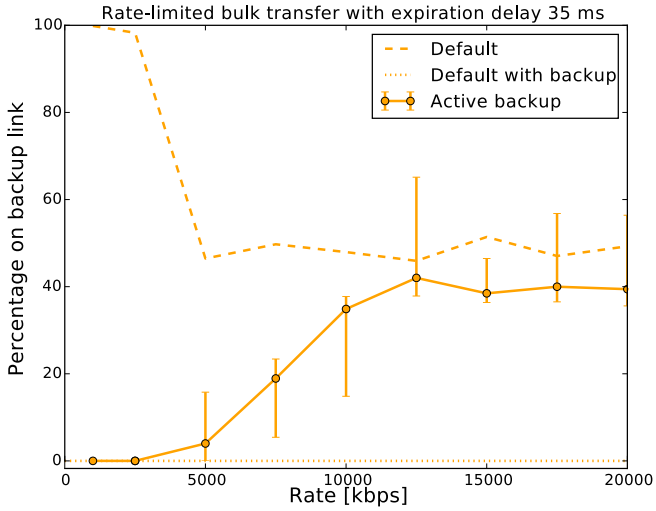


Figure 2.8: Rate-limited bulk traffic with expiration delay of 35ms. Similar trends are observed with other expiration delay values.

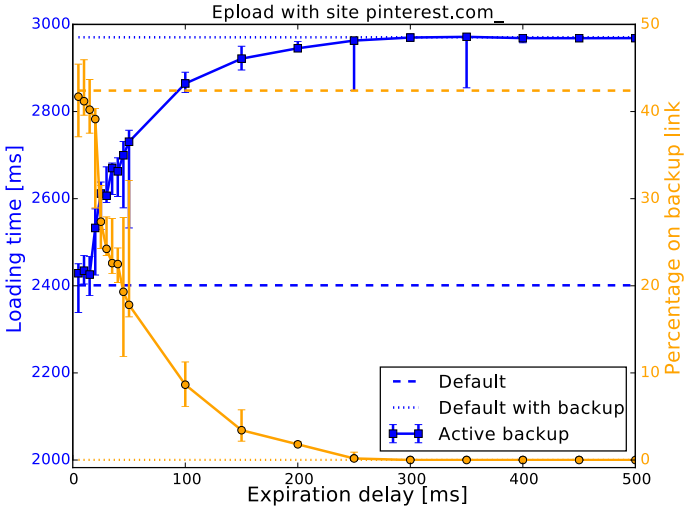


Figure 2.9: Loading `pinterest.com` using `eplоad`.

is full, the default scheduler directly uses the second path while the active backup scheduler waits until the data has expired before doing so with the backup path. The active backup scheduler is therefore useful to ensure that connections that consume a low average bandwidth will only use the main subflow and the backup subflow will only be used for heavier connections whose average bandwidth is close or larger than the capacity of the primary link.

Web page loading times are more interesting because they involve different MPTCP connections with different sizes and durations. Figure 2.9 shows that increasing the expiration delay helps to reduce the usage of the backup link. In terms of page loading time for
`pinterest.com`, the performance of the active backup scheduler lies between the default and the default backup. When the expiration delay grows, the page loading time converges to the one of the default backup. The time performance can be different depending on the web page loaded. Indeed, if the page loading only triggers short connections carrying a small amount of bytes, the backup subflow does not have the time to be used, typically because the data transfer is over before the subflow is fully established. In such cases, changing the scheduler (default, default backup, active backup) does not change anything to results. By looking closely into Figures 2.7 and 2.9, one could expect the LTE link usage for the default scheduler to be a little bit higher than 50% given the configuration of the links. Our analysis shows that this is caused by the fact that the LTE subflow is established after the primary subflow. During the establishment of the LTE subflow, the primary subflow

Config.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Path 1	0	0	0	0	0	0	0	0.1	0.3	0.5	1	1.5	2	0.1	0.3	0.5
Path 2	0	0.1	0.3	0.5	1	1.5	2	0	0	0	0	0	0	1.1	1.3	1.5

Table 2.1: Random loss percentage applied on paths.

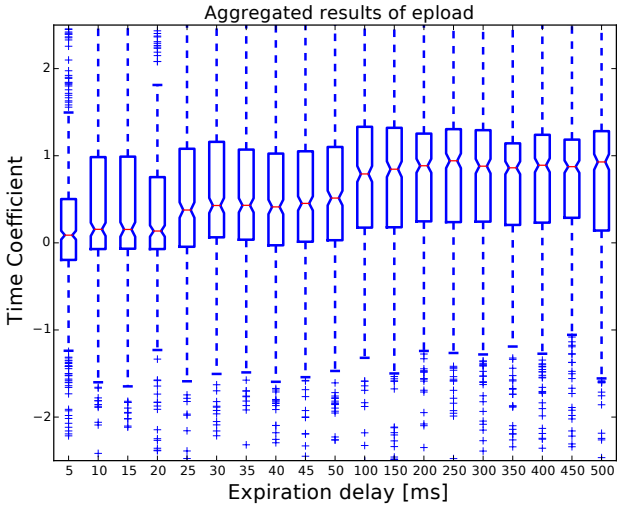


Figure 2.10: Time coefficient of `epload` with random losses (10 Mbps 30 ms RTT - 10 Mbps 20 ms RTT).

is already exchanging data, increasing it's contribution to the total amount of data exchanged. This difference will thus decrease by being amortized as the transfer size increases. This can be observed in Figure 2.9 that represents a total transfer size of 1.6MB and Figure 2.7, that represents a total transfer size of 20MB. In the first case, the usage of the LTE link represents 42.5% in the second case, it represents 47.5%.

Impact of random losses

The above measurements have considered a perfect network without any loss. We now explore the performance of the active backup scheduler when random losses occur on paths. This can give an idea of how it will behaves in wireless environments. We take the same topology as before (same delays and bandwidths) and explore 16 loss configurations described on Tab. 2.1 leading to around 10,000 tests.

To analyze those aggregated measurements, we rely on two metrics that characterize at a high level the page loading time and the usage of the backup subflow. Let T_{def}^{50} and T_{bk}^{50} be the median times to load a given web page

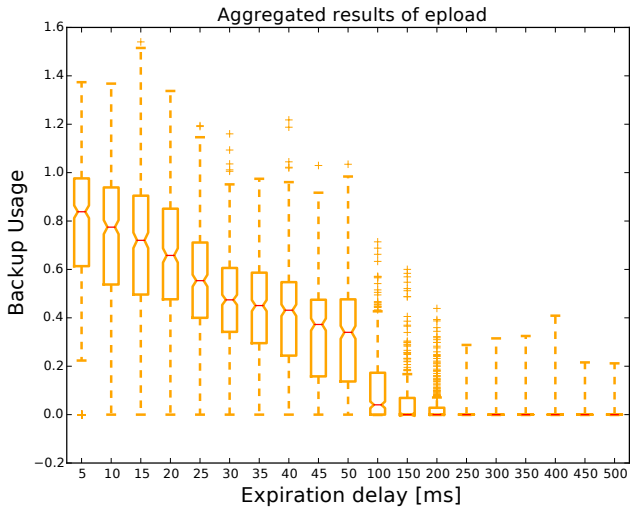


Figure 2.11: Backup usage of `epload` with random losses (10 Mbps 30 ms RTT - 10 Mbps 20 ms RTT).

with a given topology using the default and the default backup schedulers, respectively. The time coefficient TC for a test that loaded the web page in time t using the active backup scheduler is given by

$$TC = \frac{t - \min(T_{def}^{50}, T_{bk}^{50})}{\max(|T_{bk}^{50} - T_{def}^{50}|, 10^{-3})}. \tag{2.1}$$

A TC value of 0 (resp. 1) shows that the active backup scheduler loaded the web page in the same time as the quickest scheduler — typically the default one — (resp. the slowest scheduler — typically the default backup one —). In this equation, 10^{-3} is used to avoid a division by zero.

A similar metric can be defined to measure the usage of the backup subflow. Let P_{def}^{50} and P_{bk}^{50} be the median percentages of bytes transmitted on the backup path for a given web page and a given topology using the default and the default backup schedulers, respectively. The backup usage BU for a test where $p\%$ of bytes went on the backup path using the active backup scheduler is given by

$$BU = \frac{p - P_{bk}^{50}}{\max(P_{def}^{50} - P_{bk}^{50}, 10^{-3})}. \tag{2.2}$$

A BU value of 0 (resp. 1) indicates that the active backup scheduler sends the same percentage of bytes on the backup interface as the default backup (resp. default) scheduler.

Characteristic	Min	Max
Bandwidth (Mbps)	0.1	100
Round-trip-time (ms)	0	400
Loss (%)	0.0	2.0

Table 2.2: Value ranges for random topologies.

The results for the time coefficient and the backup usage are shown on Figure 2.10 and 2.11, respectively. As expected, when the expiration delay increases, the page loading time increases and the backup usage decreases. In median cases, the performances stay between the default and the default backup schedulers. This confirms the intuition that the default and default backup schedulers are extreme cases of the active backup scheduler, with expiration delays set at 0 ms and ∞ ms, respectively. In particular, using 50 ms as expiration delay provides a good trade-off for web traffic between loading time (in the middle between the default and the default backup) and backup usage (median decrease of 60% compared to the default). In these figures, the outliers are linked to the randomness of the losses. For instance, a loss on the `SYN` packet of the backup connection as a higher impact on the performances than a loss on a data packet.

Random topologies

So far, we considered the same topology with the same bandwidth and delay profiles. To see how the active backup scheduler behaves in a broader set of networks, we perform a space exploration by generating random two-paths topologies. Table 2.2 shows the range of values for the path characteristics, chosen under a uniform distribution. Notice that the buffer sizes are still equal to the bandwidth-product of their link. We run our `epload` tests on 20 two-paths topologies by considering both cases where either connection starts on path 1 and path 2 is the backup one or connection starts on path 2 and path 1 is the backup one.

Figure 2.12 shows the values of the backup usage as defined in Eq. (2.2) obtained by running more than 4,000 tests. Despite the variability of the results mainly due to the random losses, we see that the backup usage is close to 1 when the expiration delay is low and tends to 0 with the expiration delay increase, as expected. This tendency is clearer when the main path is the one having the lowest RTT. In terms of time performance, the results do not show a clear tendency, but the active backup scheduler is close of the performance of the best scheduler between the default and the default backup (depending on path characteristics), especially if the main path is the lowest RTT one.

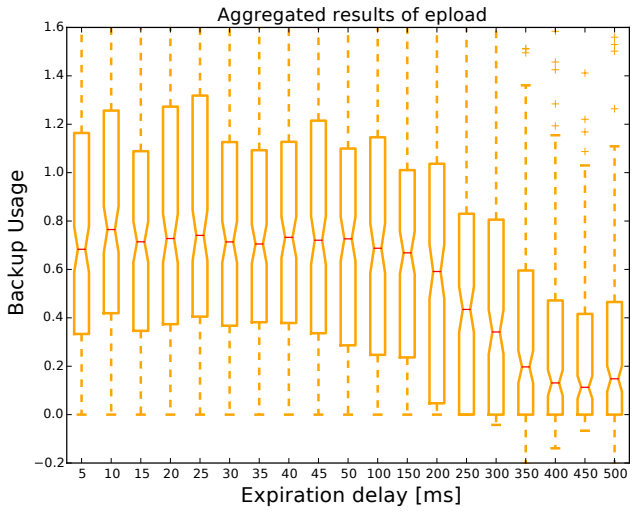


Figure 2.12: Backup usage of `epload` with random topologies.

2.2.5 Related work

Various Multipath TCP have analyzed the performance of Multipath TCP on smartphones[CT14, CLG+13, CBHB16, DNSB14]. In one of the first experimental Multipath TCP on Multipath TCP [RNBH11], Raiciu et al. explored the behavior of Multipath TCP over WiFi and 3G interfaces with bandwidth pooling, but they did not analyze the utilization of backup subflows. Later, Paasch et al. explored in [PDD+12] the behavior of Multipath TCP using the backup mode for one interface with the cellular path being used only when the WiFi subflow fails. This was the first experimental demonstration of the utilization of backup subflows. Our contribution shows that thanks to the expiration delay, it is possible to have a continuum of behaviors between bandwidth pooling and fail-over. De Coninck et al. [DCBHB16] shows in smartphone environment that backup subflows are often unused. From an energy consumption point of view, our argument in favor of the usage of the LTE subflows in backup mode to transmit data relies on [DNSB14], where Deng et al. showed that the "Tail Energy" phenomenon considerably limits the energy saving when using cellular as a backup interface.

Regarding the scheduling algorithms, Paasch et al. [PFAB14] designed and evaluated a Round-Robin scheduler and two variants of the default (Lowest RTT) scheduler and used experimental design to demonstrate the effects caused by bad scheduling decisions. Ferlin et al. proposed BLEST [FAMB16], a scheduler aimed at mitigating the effects of Head-Of-line blocking while using heterogeneous paths. In [OL15], Oh et al. proposed another scheduler

that tackles the same problem. Arzani [AGC+14a] showed that the choice of the scheduling policy also depends on path characteristics. The Socket intent approach[SEKF13] allows the application to inform the network stack about several informations (how the traffic may look like, the tolerance of the application...) about its communication pattern.

2.2.6 Discussion

In this section, we have proposed and implemented a new way of handling backup subflows in Multipath TCP. Our solution provides a trade-off between packet time delivery and the utilization of the backup interface. Instead of transmitting over the backup subflows only when the primary ones fail, we measure the time spent by the data in the Multipath TCP stack and use the backup subflow as soon as the data has been delayed by more than a configured time. We implement this as an extension of the Multipath TCP scheduler that is configured with an expiration delay on a per-connection basis. Our measurements show that this keeps most of the traffic on the primary subflows for interactive applications such as web browsing.

Our solution could be part of a higher level API such as the one proposed for socket intents [SEKF13] where the application specifies its high level expectations and the planned utilization of the connection to allow the stack to automatically configure parameters such as our expiration delay to match the user's needs.

2.3 Trusted resource pooling with Multipath TCP

2.3.1 Introduction

Smartphones and tablets are becoming one of the most widely used devices to access the Internet. Today's smartphones are equipped with several wireless interfaces (WiFi, 4G, Bluetooth, ...). Faced with a huge growth of the data traffic [ER13, HQG+13], mobile network operators are exploring alternatives to 4G to provide Internet connectivity.

Researchers have explored the interactions between WiFi and 4G in the past. Several studies have demonstrated that there are performance and cost benefits in offloading data traffic to the WiFi network [LLY+13]. These findings encouraged network operators to roll out large WiFi networks.

From a pure cost viewpoint, users and network operators could wish to offload their traffic onto WiFi networks. However, WiFi networks also have some drawbacks. First, a WiFi network may be much slower than 4G, in particular when many users are attached to a low bandwidth broadband link. Second, using WiFi may expose the users to more types of attacks and secu-

rity problems than 4G networks. Cellular networks are typically controlled by the network operators and it is difficult for attackers to capture or inject packets inside these networks. On the other hand, WiFi started as a completely open technology and there are still many open access points where all data packets can be easily eavesdropped. Users attached to open WiFi networks are vulnerable to a wide range of attacks such as the Firesheep[1] Firefox extension that allows to hijack HTTP sessions. Furthermore, many ADSL/cable routers that provide WiFi access have often been the target of attacks, some having compromised hundreds of thousands of routers (see e.g. [MSM13, Goo13]). Once compromised, such a WiFi router can easily mount various types of man in the middle attacks.

Multipath TCP was designed with resource pooling in mind [WHB08] and aims at distributing data fairly over different interfaces. In this section, we show that equally distributing data over different networks is not sufficient when these networks have different properties from a security viewpoint. We extend Multipath TCP to take into account the trust level of each network interface and evaluate the performance of our implementation.

This section is organized as follows. In Section 2.3.2, we explain how Multipath TCP and our Linux implementation can be extended to take this level of trust into account. Section 2.3.3 explains how Multipath TCP can benefit from the secure handshake used by protocols such as SSL or TLS. Finally, we discuss our findings in Section 2.3.4.

2.3.2 (Un)Trusted network interfaces

Multipath TCP was designed to be no less secure that regular TCP [BPG+14]. In contrast with TCP extensions such as TCPCrypt [BHH+10],
TCP-AO [TMB10] or QUIC [IT19], Multipath TCP does not encrypt or authenticate the segments that it sends. The only "secure" mechanism used in Multipath TCP is the authentication of the subflows. As explained in Section 1.4.3, the client and the server exchange 64 bits keys in clear during the initial three-way handshake. These keys are then used later to authenticate the additional subflows with an HMAC computation.

If an attacker can eavesdrop a key exchanged during the initial three-way handshake, he/she can easily add a new subflow to an existing Multipath TCP connection and mount some attacks by injecting or collecting data. The Multipath TCP specification [FRHB13] and the current implementation in the Linux kernel [RPB+12] does not propose a solution to this problem. Adding a strong key exchange scheme in Multipath TCP as defined in [FRHB13] appears impossible given the limited space available in the TCP options [2].

[1] See `http://codebutler.github.io/firesheep/`

[2] TCP options cannot be longer than 40 bytes and the option space in the initial SYN seg-

Given that usually the user can trust its mobile network operator (and 3G/4G includes protocols to verify the mobile network), we propose to extend Multipath TCP by distinguishing two types of network interfaces :

- **Trusted interface**. An interface is considered to be **trusted** when the user can expect that passive and active eavesdropping will be impossible on this interface given its nature (e.g. physical wire) or due to the utilization of encryption techniques (e.g IPSec).
- **Untrusted interface**. An interface is consider to be **untrusted** if attacker can easily eavesdrop packets.

In our implementation, we extend the interfaces table in the Linux kernel with one bit per interface that indicates its trust level. This trust level will typically be automatically configured by the connection manager, that controls the utilization of the network interfaces. We expect that wired interfaces such as Ethernet could be considered as trusted by default. However, some companies could prefer to consider that only the company's Ethernet is trusted and rely on 802.1x to verify that the device is attached to the company network. Virtual Private Network solutions built with IPSec, SSL/TLS or DTLS usually provide a virtual network interface. Such interfaces will be considered as trusted by the connection manager. For wireless networks, we expect that 3G and 4G networks will be considered to be trusted given the utilization of link layer encryption to secure the wireless channel. Mobile network operators often install tailored connection managers on the smartphones that they sell. This connection manager could easily recognize the operator's networks and consider them to be trusted. For WiFi networks, the level of trust could depend on the use of link-layer encryption (e.g. WPA2 could be considered trusted while WEP would not be) and also on the utilization of 802.1x (e.g. a corporate WiFi network using EAP-TTLS would be considered to be trusted once the network certificate has been validated).

Protecting the initial handshake

With the level of trust of each interface in mind, we need to reconsider how the TCP/IP stack uses the available interfaces. In the existing Linux TCP/IP implementation, one interface is considered to be preferred. On smartphones, this is usually the WiFi interface when active given that WiFi usually[3] provides a lower delay and higher throughput than cellular networks.

Our first modification to the Multipath TCP implementation is to take the trust level of the interfaces into account when creating a Multipath TCP

ment is already almost full.

[3]This might change in cellular networks supporting 4G/LTE that provide lower delays [HQG+13].

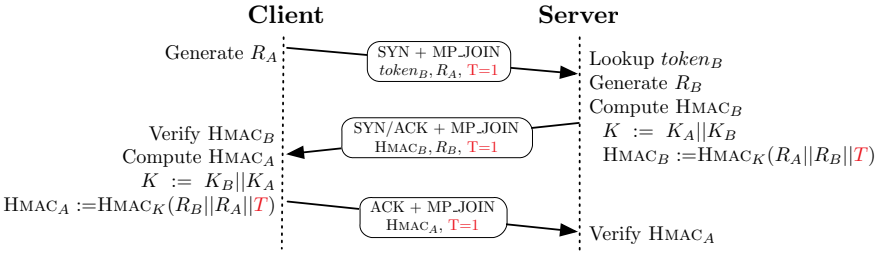


Figure 2.13: Modified subflow establishment.

connection. When a client opens a Multipath TCP connection, our implementation forces the transmission of the first `SYN` packet over a trusted interface. This protects the random keys that are included in the `MP_CAPABLE` option against passive eavesdroppers. Once the initial subflow has been created over a trusted interface, additional subflows can be established over the other available interfaces.

Configuring the trust level of each interface on the client is necessary but unfortunately not sufficient. Since both the client and the server will exchange data, there must be a way for the client to reliably inform the server about the trust level of the different subflows. We first assume that the initial subflow is always established over a trusted interface. We further assume that the server only uses trusted interfaces. The second subflow could be established over a trusted or an untrusted interface. To convey the trust level of the interface used on the client to create the subflow, we extend the `MP_JOIN` option with the T bit. When set, this bit indicates that the subflow is created on an trusted interface on the client. Otherwise the interface (and thus the subflow) is considered to be untrusted. To prevent active attacks from e.g. a rogue WiFi access point, the value of the T bit is included in the HMAC computation used to authenticate the subflow.

Protecting the data

Protecting the initial handshake does not protect the data. This is particularly important with plain text protocols such as HTTP, FTP, NFS or SMTP.

On smartphones, HTTP/HTTPS remains the most widely used application [ER13, HQG+13, FLM+10]. In the recent years, the usage of HTTP has declined in favor of HTTPS. However, HTTP is surprisingly still largely used [FBK+17, Gro19]. Multipath TCP cannot encrypt the data to protect it, but applications can decide to send sensitive data only over trusted interfaces. For HTTP, we need to distinguish between the HTTP headers (request and response) and the web objects that are exchanged. The HTTP headers often

```
...
/*This message must be sent on a trusted interface*/
msg.msg_iov->iov_base = secure_content;
msg.msg_iov->iov_len = length_of_secure_content;
sendmsg(sockfd,&msg,MPTCP_TRUSTED);

/*This message may be sent on any interface*/
msg.msg_iov->iov_base = unsecure_content;
msg.msg_iov->iov_len = length_of_unsecure_content;
sendmsg(sockfd,&msg,0);
...
```

Figure 2.14: Using `sendmsg` to force the utilization of trusted subflows

contain sensitive data such as cookies or other forms of session identifiers [VB10].

To enable applications to request the transmission of sensitive data only over trusted interfaces, we extend the Multipath TCP implementation by overloading the `sendmsg` system call. This system call allows to transmit data on a socket while supporting scatter-gather function. This system call allows the application to specify some flags when sending data. Our implementation defines the new `MPTCP_TRUSTED` flag. When this flag is set, the buffer passed through the `sendmsg` system call must only be transmitted over trusted interfaces. The `send` and `sendto` system calls can be modified in a similar way to support the `MPTCP_TRUSTED` flag.

When an application calls `sendmsg` and requires the transmission of data over a trusted interface, the stack iterates over the available subflows to find one that is trusted and has enough space in its congestion window to send the data. If necessary, the transmission of the data is delayed until a trusted subflow becomes available. Although our implementation forces the data sent with the modified `sendmsg` system call to be transmitted over a trusted subflow, it does not force this data to be transmitted in a (sequence of) independent packet(s). Other data may be attached before or after these packets.

Performance evaluation

In this section, we experimentally evaluate the performance impact of forcing Multipath TCP to send the HTTP headers and responses only over a trusted interface. We use a simple network composed of one client, one router and one server. The client is a standard Linux server because it is easier to automate measurements on such a server than on a smartphone, but our implementation also works on Android smartphones. The router is also a Linux PC that uses the `tc` software to emulate bandwidth and delays. For our evaluation, the bandwidth is set to 5, 10, 20 and 50 Mbps on the untrusted interface, emulating a WiFi access point, and 1, 5, 10, 20 Mbps on the trusted interface,

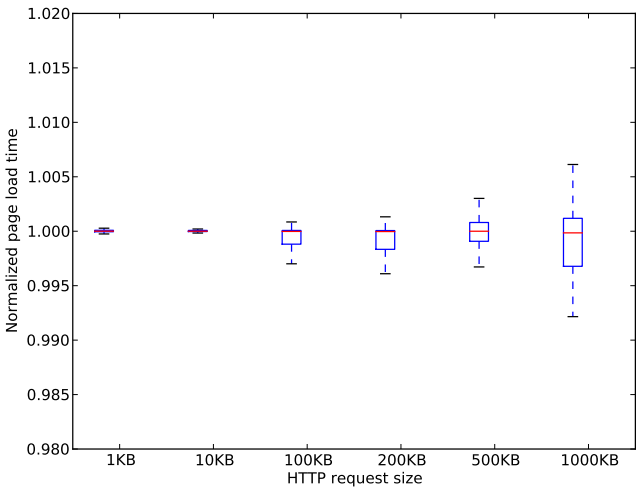
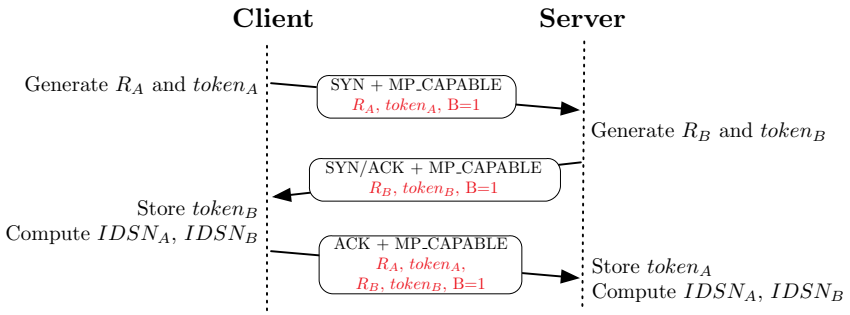


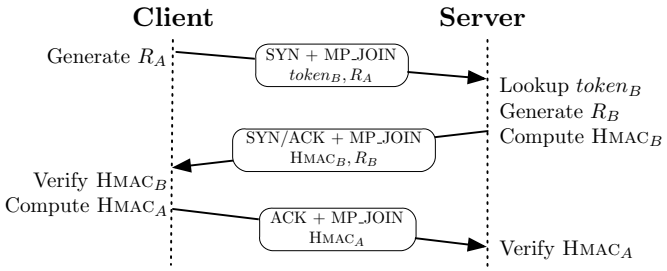
Figure 2.15: Boxplots show the ratio of page-load time between trusted and regular Multipath TCP. It can be seen that there is no significant difference between both.

representing a cellular interface. We set delays of 10, 100 and 200 msec on the trusted and untrusted interface. Our measurement scripts then iterate over all possible combinations of bandwidth and delays for the cellular and WiFi paths. Iterating over multiple conditions allows us to have a better view of the performance of Multipath TCP. In some combinations, WiFi is faster than cellular, in others this is the opposite. This reflects real-world deployments where both situations could happen.

We consider HTTP requests for objects of 1, 10, 100, 200, 500 and 1000 Kbytes. Each HTTP measurement is repeated ten times and for each bandwidth/delay combination we measure the average HTTP page load time. The client and the server use `sendmsg` to always send the HTTP headers over the trusted, "cellular" interface. Figure 2.15 compares the download times between Multipath TCP with a trusted interface and regular Multipath TCP. We divide the page-load time of the former by the page-load time of the latter. It is apparent in the graph that indeed, within our large range of environments, the difference between both is negligible as the boxplots are centered around 1. There is no significant performance decrease due to the utilization of trusted interfaces. With larger filesizes the spread of the page-load times is a bit larger and thus a minor variance can be seen. Increasing the number of repetitions would reduce this variance.



(a) Initial subflow establishment.



(b) Additional subflows establishment.

Figure 2.16: MPTCP external keys: MPTCP's initial handshake is simplified, the B-bit is set to 1. To establish additional subflows, a similar security mechanism as standard Multipath TCP is used.

2.3.3 Multipath TCP and SSL/TLS

Security-critical traffic is often encrypted by using protocols like SSL/TLS. There is a growing motivation to use secure protocols like SSL to transfer data as demonstrated by the *"Always on SSL"*-call from the *Online Trust Alliance* (OTA)[4] or [JB08].

If SSL/TLS is used, we can also prevent hijacking attacks, as the attacker needs to know the shared secret of the SSL-session in order to inject traffic. SSL secures each record with a Message Authentication Code (MAC), using the shared secret of the SSL-session. If the MAC of a received record is incorrect, SSL destroys the connection with an error alert (*bad_record_mac*) [Res01].

This prevents the attacker from injecting data inside an SSL session, but a successful packet injection would cause a denial of service given the release

[4] `https://otalliance.org/resources/AOSSL/index.html`

of the TCP connection and the SSL session. With Multipath TCP, an attacker that has eavesdropped the initial handshake but cannot send spoofed packets is able to create a new subflow and inject data that will be delivered by Multipath TCP to the SSL layer.

To prevent this attack, we propose to extend the Multipath TCP handshake to derivate the Multipath TCP keys from the shared secret that has been securely negotiated by SSL to authenticate the additional subflows. As the shared secret is not sent in plaintext over the network, an attacker is prevented from creating new subflows and authenticating himself.

We extend the Multipath TCP protocol and its Linux kernel implementation to enable the SSL library to securely negotiate the authentication key used by Multipath TCP.

When an application opens a new socket it informs Multipath TCP that a key for authenticating new subflows will be provided by the application. For this, we define a new socket option: `MPTCP_ENABLE_APP_KEY`. When sending the MP_CAPABLE option, the B-bit is set to 1, (see [FRHB13] for more details about the reserved flags in the MP_CAPABLE option), to signal to the peer that external keys will be used for the additional subflows. The initial subflow's handshake can thus be simplified. Since the keys are computed by SSL they do not need to be included in clear in the MP_CAPABLE option. Still, a random number and the token to identify the Multipath TCP session as well as generate the initial Data Sequence Number (IDSN) are included in the MP_CAPABLE. We refer the interested reader to [PB12] for more details on the format of the MP_CAPABLE option.

The secure handshake used by SSL and TLS concludes with a Master-Secret. SSL and TLS use a Key Derivation Function (KDF) to derive the encryption and authentication keys that are used to protect the data exchanged over the SSL session. This KDF is extended to derive two new keys that will be used by Multipath TCP. One key is used on the client to authenticate the subflows that it initiates and the other key is used by the server. Since Multipath TCP is implemented in the kernel and SSL resides in a library in userspace the library must provide the keys to the kernel. For this, we extend the Multipath TCP implementation in the Linux kernel with a new socket option : `MPTCP_KEY`. This option enables the application to pass the client and server keys to Multipath TCP in the kernel.

Our modified handshake [PB12] for the new subflows is described in Figure 2.16. There may of course be situations, where the client is using the B-bit, but the server not. In this case, the server does not respond with the B-bit set. Upon reception of the SYN/ACK, it is up to the client's policy to decide on how to react. It can either fallback to regular Multipath TCP, fallback to regular TCP or reset the connection and start from scratch a regular TCP connection, etc.

Performance evaluation

We implemented the proposed solution in the Linux kernel and measured how long it takes to establish a second subflow, compared to standard Multipath TCP. For this, we slightly modified OpenSSL. A similar modification could be performed on any other application-level protocol that negotiates a shared secret such as SSH.

The current Multipath TCP implementation in the Linux kernel opens a second subflow after 2 round-trip-times. This delay is mandatory to verify that there are no middleboxes on the path that remove TCP options [RPB+12]. With external keys, an additional subflow can only be established after the end of the SSL/TLS handshake.

Type	Data Center Delay	Internet-like Delay
Standard MPTCP	352 ± 2 µs	40.372 ± 0.002 ms
MPTCP ext. keys	1910 ± 13 µs	61.934 ± 0.02 ms

Table 2.3: Using the external keys delays the establishment of an additional subflow by one RTT in the realistic *Internet-like*-scenario.

We evaluate this in our lab testbed with two different scenarios (Table 2.3). We first consider that the servers are directly connected to the same switch. In this environment, the second subflow is established within 352 ± 2 µs by the standard Multipath TCP implementation. When external keys are used with OpenSSL, the delay grows to 1910 ± 13 µs. Note that this includes the secure handshake and the derivation of the keys. This environment is a worst case scenario since all server interfaces are usually trusted and external keys are not required inside a datacenter.

We then consider an Internet like scenario and emulate a 20 ms delay between the hosts. In this case, it becomes apparent, that standard Multipath TCP consumes two round-trip-times before establishing the second subflow. With external keys, the delay increases to three round-trip-times. We argue that this additional delay is not of a big concern. In *Internet-like* scenarios, the second subflow is delayed by only one round-trip-time. Moreover, with SSL no data is sent until the shared secrets have been exchanged by SSL. The second subflow can be established as soon as SSL starts sending the application's data.

Using trusted interfaces

SSL and TLS are much more secure than plaintext protocols, but they are far from perfect and there are situations where the Multipath TCP extensions

discussed in the previous section could also be applied with SSL/TLS. SSL and TLS rely on server certificates to authenticate the servers. The validation of these certificates has always been a weak point in client implementations and has enabled researchers to implement software that can be used on WiFi access points to intercept, modify, replay and save traffic presumably secured by SSL/TLS[5]. In 2014, several bugs in the certificate validation logic of popular SSL/TLS implementations (GnuTLS and Apple) have been identified and exploited by such interception software. Given the importance of the SSL/TLS certificates, some smartphone users might want to always use trusted interfaces for the initial TLS/SSL handshake and the certificate exchange.

The handshake is not the only vulnerable part in the SSL/TLS protocol from a cryptographic viewpoint. TLS/SSL supports different combinations of encryption and authentication algorithms. Some are known to be weaker than others, e.g. due to the length of the encryption keys that are used. While RC4 was one of the most widely used encryption algorithms on web servers in 2015, it is unfortunately still used by some servers. A recent survey[6] reveals that among 139,154 analyzed web servers supporting SSL/TLS, 13% of them still support RC4. This is despite serious concerns from the cryptography community that considers RC4 to be broken [ABP13]. In particular, [ABP13] showed that the first hundred bytes of the encrypted stream are more vulnerable to cryptanalysis than the remaining bytes. If a client still needs to interact with an SSL/TLS server that uses RC4, it could force the utilization of trusted interfaces for the SSL/TLS handshake and the vulnerable bytes in the beginning of the stream.

2.3.4 Discussion

In this section, we have shown that traffic offload solutions need to also consider the trustiness of the networks when offloading traffic. Cellular network vendors have proposed IPSec-based solutions to deal with this security issue [San12, dlOBC+11]. With Multipath TCP, a different approach is possible.

We have first proposed to encode the trust level of each network in the interfaces table and implemented this extension in the Linux kernel. Then, we have analyzed how Multipath TCP should deal with trusted and untrusted interfaces when used with both plaintext and secure protocols.

With plaintext protocols such as HTTP, that are unfortunately still used, our solution is to force Multipath TCP to always use a trusted interface to establish the connection (where the authentication keys are exchanged in clear) and then establish subflows over untrusted interfaces. We have also extended the `sendmsg` system call to allow an application to force the transmission of

[5] See e.g. `http://mitmproxy.org`.

[6] See `https://www.ssllabs.com/ssl-pulse/`

sensitive data only over trusted interfaces. These two extensions have been implemented in the Linux kernel and our measurements indicate that they have a very small impact on the raw performance of Multipath TCP.

With secure protocols such as TLS/SSL, the situation is different. The cryptographic techniques used by such protocols can protect the data. However, exchanging the keys that Multipath TCP uses to authenticate the subflows in clear during the initial handshake could open a new form of denial of service attack. Such attacks can be prevented by deriving the Multipath TCP key from the secure SSL/TLS handshake. We extended our Multipath TCP implementation in the Linux kernel to support this new feature and evaluate its performance. SSL/TLS sometimes uses less secure cryptographic algorithms such as RC4. In this case forcing the utilization of a trusted network for the beginning of the encrypted data stream could be a good countermeasure.

Although our solution has been applied for Multipath TCP, it could also be useful for other protocols. The Domain Name System (DNS) is another example of a protocol that is sensitive from both security and privacy viewpoint [ZHH+15]. The DNS resolver on smartphones could be extended to only use trusted interfaces when sending DNS queries.

Chapter 3

SRv6Pipes: enabling in-network bytestream functions

3.1 Introduction

Middleboxes play an important role in today's enterprise and datacenter networks. In addition to the traditional switches and routers, enterprise networks contain other devices that forward, inspect, modify or control packets. There is a wide variety of middleboxes [CB02], ranging from simple NAT, IP firewalls, various forms of Deep Packet Inspection, TCP Performance Enhancing Proxies (PEP), load balancers, Application Level Gateways (ALG), proxies, caches, edge servers, etc. Measurement studies have shown that some networks have deployed as many middleboxes as the number of traditional routers [SHS+12].

Those middleboxes were not part of the original TCP/IP architecture. They are typically deployed by either placing the middleboxes on the path of the traffic that needs to be handled, *e.g.*, on the link between two adjacent routers, or by using specific routing configurations to force some packets to pass through a particular middlebox. These two deployment approaches are fragile and can cause failures that are hard to diagnose and correct in large networks. Pothraju and Jain have shown in [PJ13] that middlebox failures are significant and that many of them belong to a grey zone, *i.e.*, they cause link flapping or connectivity errors that are difficult to debug and impact the end-to-end traffic. Researchers and vendors have proposed Network Function Virtualization (NFV) [JB16] and Service Function Chaining (SFC) [HP15] to solve some of the problems caused by middleboxes.

In a nutshell, the NFV paradigm argues that all network functions should

be virtualised and executed on commodity hardware instead of requiring specific devices. On the other hand, SFC [HP15] proposes to support chains of network functions which can be applied to the packets exchanged between communicating hosts. Several realisations for SFC are being discussed within the IETF. The SFC working group is developing the Network Service Header [QEP17]. This new header can be used to implement service chains and replaces already deployed proprietary solutions. Another approach is to leverage the extensibility of IPv6. Given the global deployment of IPv6 [NG16], several large enterprises have already announced plans to migrate their internal network or their datacenters to IPv6-only to avoid the burden of managing two different networking stacks [For17]. In addition to having a larger addressing space than IPv4, IPv6 provides several interesting features to support middleboxes in enterprise and datacenter networks. One of these is the native support for Segment Routing [FNP+15, FDP+19].

In this chapter, we demonstrate the benefits that the IPv6 Segment Routing (SRv6) architecture can bring to support middleboxes in enterprise and datacenter networks. With SRv6, middleboxes can be exposed in the architecture and visible end-to-end. This significantly improves the manageability of the network and the detection of failures while enabling new use cases where applications can select to use specific middleboxes for some end-to-end flows. This chapter is organized as follows. In Section 3.2, we describe some use cases that can benefit from middleboxes. In Section 3.3, we present *SRv6Pipes*, a modular SRv6-based architecture to support arbitrary in-network Virtual Functions, that can be applied on bytestreams and chained together. In Section 3.4, we detail a prototype implementation of our architecture, running on Linux. In Section 3.5, we demonstrate the feasibility of our approach and evaluate the performance of our prototype through various tests and microbenchmarks. Finally, we cover some related work in Section 3.6 and conclude in Section 3.7. Future work is discussed in Section 3.8.

3.2 Use Cases

Middleboxes can perform two different types of network functions: *per-packet* and *per-bytestream*. The *per-packet* functions operate on a per-packet basis. They include Network Address Translation and simple firewalls. These functions typically operate on the network and sometimes transport headers. The *per-bytestream* functions are more complex, but also more useful. These functions operate on the payload of the TCP packets. For example, firewalls and Intrusion Detection Systems (IDS) need to match patterns in the packet payload while transparent compression and/or encryption need to modify the payload of TCP packets. Such functions need to at least reorder the received

TCP packets but often need to include an almost complete TCP implementation. We describe some of these *per-bytestream* functions in more details in this section.

3.2.1 Application-level Firewalling

To cope with various forms of packet reordering, application-level firewalls and Intrusion Detection/Prevention Systems need to at least normalize the received packets [KHP01] before processing them. Another approach is to use a transparent TCP proxy on the firewall to terminate the TCP connection and let the firewall/IDS process the reassembled payload. An end-to-end connection would thus be composed of two sub-connections: one between the client and the middlebox and another one between the middlebox and the server. Network operators often configure access lists to associate IP prefixes to some security checks performed by the IDS. For example, in a University network, student laptops would be subject to different policies than servers.

3.2.2 Multipath TCP Proxies

Presentend in Chapter 1.4, Multipath TCP [FRHB13] (MPTCP) enables hosts to send packets belonging to one connection over different paths. One of the benefits of MPTCP is that it allows to aggregate the bandwidth of multiple connections. This enables, *e.g.*, network operators to bond xDSL and LTE networks to better serve rural areas [BS16]. However, MPTCP being an end-to-end protocol, the client and the server require MPTCP-enabled kernels. To leverage the benefits of MPTCP without modifying the client or server network stacks, operators started developing MPTCP-aware proxies [BS16, BBG+19] to convert regular TCP to MPTCP and conversely.

To allow the bundling of xDSL and LTE, an NFV deployment could be leveraged to implement the same behavior, by placing a proxy in the CPE to convert regular TCP to MPTCP and a second proxy in the operator's network to convert MPTCP to regular TCP. This would allow non-MPTCP clients and servers to use different networks simultaneously.

In practice, network operators could want to support different services on the same proxy, e.g. *(i)* a business proxy that always maximizes bandwidth for business customers, *(ii)* a low-cost proxy that only uses the LTE network when the xDSL network is fully utilized or *(iii)* a gaming proxy that always uses the network that provides the lowest delay. Such proxies can be deployed by tuning the packet scheduler and the path manager of Multipath TCP implementations.

3.2.3 Load Balancing

Load balancing is a fundamental requirement of many networks. Two main variants are possible. The first variant is in the presence of multiple physical servers providing a single service. The downstream proxy of the segmented path could be configured to perform load balancing amongst these servers. To select the server, the proxy would act in an LVS-like fashion [LVS] and select one of the servers using, *e.g.*, a Deficit Weighted Round Robin algorithm. The transport-layer load balancing can easily be upgraded to an application-layer load-balancing, by using some parameters received by the proxy, or by inspecting the content of the payload to select the backend server that fits the application protocol.

The second variant consists in balancing the load across several proxies. If one of the Virtual Functions is performing computationally intensive operations, it can rapidly become a bottleneck. To prevent this, an upstream function can load-balance the connections across multiple identical Virtual Functions residing on different proxies. A simple implementation of this variant can be achieved by statically configuring the load-balancing proxy with the available proxies for a given function or to use configuration parameters received by the proxy. Another option is to leverage an Opaque Container TLV [FDP+19], however the specifications discourage the usage of TLVs for intermediate nodes.

3.2.4 Multimedia transcoding

Multimedia transcoding has been a research topic for a long time [AMK98, XLS05]. Since, it has been widely deployed by companies like Amazon [amazontranscode]. In this context, a proxy placed between the client and the server that hosts the multimedia file can be used to transcode the multimedia file hosted on the server into a format compatible with the client. This allows to distribute the computation intensive task of transcoding the content over several proxies, while the server simply hosts the original files. In this setup, parameters could be passed to the proxy to specify for instance the maximum bitrate that a client is entitled to (based on technical or subscription limitations), the maximum number of streams allowed for this client or the type of content authorized for this client.

3.3 Architecture

Middleboxes and other in-network functions are installed, configured, and managed by network administrators according to business (e.g. security regulations impose the utilisation firewalls) and technical (e.g. performance issues

force the utilisation of performance enhancing proxies, or addressing issues force the utilisation of NAT) needs. Usually, network administrators impose the utilisation of specific network functions by configuring routing policies or placing physical boxes on links that carry specific traffic (e.g. firewalls are often attached to egress links). This is both cumbersome and costly since all possible links must be covered by each intended network function.

Like NFV, our architecture assumes that network functions are software modules which can be executed anywhere in the network. A firewall function that only needs to process the external flows does not need to be installed on the egress router, it can be executed on any server or router inside an enterprise network. Each network function is identified by an IPv6 prefix which is advertised by the equipment hosting the function (see Section 3.3.3). For redundancy or load-balancing, the same function can be hosted on different equipments in the network.

To understand the different elements of our architecture, let us consider a simple scenario. A client host needs to open a TCP connection towards a remote server. The network administrator has decided that the packets belonging to such a connection must be processed by two network functions: *(i)* a stateless firewall which blocks prohibited ports and *(ii)* a DPI which inspects all external TCP connections. Three elements of our architecture are used to support this sequence of network functions in enterprise networks.

The first element is IPv6 Segment Routing (SRv6) [FPG+18]. Our architecture uses the SRv6 header (`SRH`) to enforce an end-to-end path between the client and the server which passes through the two equipments hosting the mandatory networking functions. We describe SRv6 in more details in Section 3.3.1.

The second element of our architecture is how the client learns the `SRH` suitable to reach a given destination. For this, we modify the enterprise DNS resolver. Instead of simply resolving names into addresses, our DNS resolver acts as a controller [Leb17, LJC+18] which has been configured by the network administrator with various network policies. When a client sends a DNS request to the resolver, it replies with the intended response and additional records which contain the `SRH` that the client has to apply to reach the specified addresses.

Thanks to the `SRH` which is attached by the client, all the packets belonging to the TCP connection will pass through the stateless firewall and the DPI. Consider now what happens if some packets are lost and need to be retransmitted. The stateless firewall is not affected since it only processes the network and transport headers that are present in each packet. On the other hand, the DPI function needs to include a TCP implementation to be able to detect out-of-order packets or other TCP artifacts. Instead of requiring each network function to include a TCP implementation, our architec-

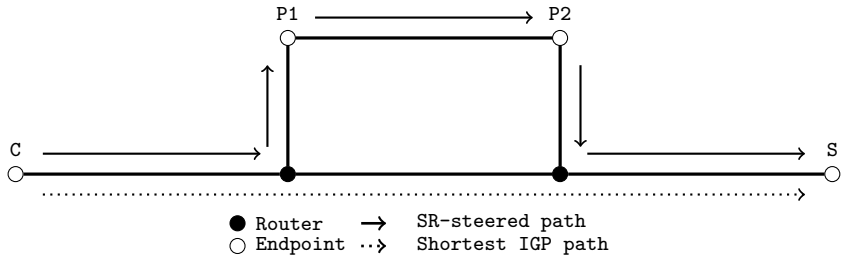


Figure 3.1: Traffic steering through two off-path network functions `P1` and `P2` (*e.g.*, firewall and IDS).

ture leverages the TCP stack that is already present in the Linux kernel. Each equipment that hosts a network function uses a transparent TCP proxy that transparently terminates the TCP connections and exposes bytestreams to the network functions as in FlowOS [BAM13]. This greatly simplifies the implementation of per-bytestream network functions

3.3.1 IPv6 Segment Routing

In SRv6Pipes, we leverage the ability of the IPv6 Segment Routing architecture described in Chapter 1.5 to steer packets through arbitrary network path to steer TCP flows through arbitrary network functions. In this architecture, a Segment Routing endpoint becomes a node that performs an operation on the bytestream. See Figure 3.1 for an illustration. Consider that client `C` establishes a connection to server `S`, with two intermediate network functions at `P1` and `P2`. To realise that, the client attaches a `SRH` to its packets, containing three segments. The first two segments represent the functions to be executed at resp. `P1` and `P2`. The third segment is the address of `S`. When the packets are transiting between `C` and `P1`, and between `P1` and `P2`, their IPv6 destination address is thus the address of the function to execute at the corresponding proxy. Between `P2` and `S`, the segment pointer is decremented to zero and the IPv6 destination address of the packets is the address of `S`.

3.3.2 Transparent TCP Proxy

The proxy is the core component of our architecture to support per-bytestream network functions. It is transparent at the network layer, meaning that even if the proxy actually terminates the TCP connection with the client, the destination server will receive packets coming from the client's IP address, and not from the proxy's IP address. The transparent proxy is placed on the path thanks to the IPv6 Segment Routing Header (`SRH`) [FDP+19]. It intercepts each new connection that matches a given pattern (*e.g.*, a destination port)

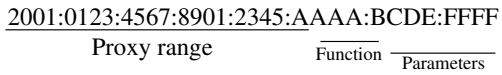


Figure 3.2: IPv6 address encoding.

and terminates it. Then, the proxy establishes a downstream connection to the next segment specified in the SRH of the inbound connection. When the proxy receives data from the client, it applies a transformation function (*i.e.*, the Virtual Function) to the received data and forwards the result on its outbound connection to the next segment of the path. This process is then repeated until reaching the final destination of the path. A consequence of this architecture is that the proxy must be able to process a clear-text stream. If the stream is encrypted, the proxy will not be able to apply meaningful transformation functions. In this chapter, we consider that the end-to-end stream from the client to the final server is not encrypted. As future work, one could explore encrypted end-to-end streams with the proxies acting as TLS termination points. Additionally, we do not consider the QUIC [IT19] protocol, as it leverages UDP rather than TCP.

3.3.3 Encoding Functions and Parameters

As shown in Section 3.2, some parameters can be passed to the per-bytestream function. Such parameters are usually specified in the proxy configuration files. However, such configurations can be large and complex if some parameters can change on a per connection basis. Consider for example a first proxy that encrypts the payload and a second that decrypts it. Those encryption/decryption proxies would have to be configured with the encryption/decryption keys for each flow. A possible approach would be to define one key per host or set of hosts. A better approach is to configure a set of keys on the proxies and associate each key with a unique identifier. When a connection starts, the encryption proxy selects a random key and places the identifier of the chosen key in the SRH towards the decryption proxy.

To enable such a granularity in the choice of transformation functions and parameters, we leverage the large addressing space available in IPv6. Each proxy announces one or more IPv6 prefixes that correspond to the Virtual Functions it hosts. Within the host part of the prefixes, we allocate a given amount of bits to encode the identifier of the function to apply as proposed in [FGL+19]. The remaining low order bits are used to specify parameters of the virtual function such as the decryption key in the above example. Consider Figure 3.2 for an illustration. The proxies announce /80 prefixes. The first 80 bits of the address thus specify the proxy to traverse. The 16 following bits identify the function to apply to the payload, and the low-order 32 bits contain

the parameters of these functions. The SRH then contains a list of proxies with their respective functions and parameters. This approach allows clients to use any combination of function/parameter available in the network.

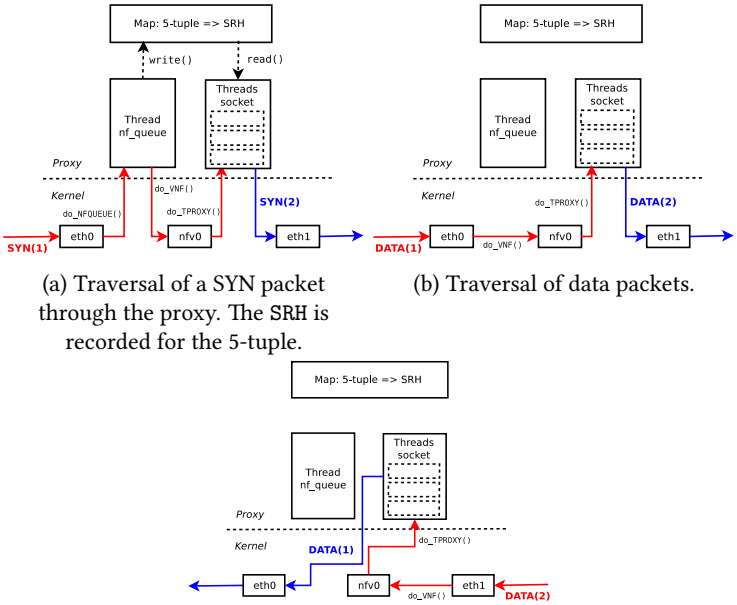
Consider the network described in Figure 3.1. In this network, the client might require to encrypt the traffic between P1 and P2. In our architecture, the client will use the function bits of the address of P1 to specify the identifier of the encrypt function, and the parameters bits to specify the identifier of an encryption key. The same will be done in the address of P2 with the decrypt function. This allows to have different encryption keys for different connections without having to store a configuration for each connection in the proxy. The processing of the return traffic is discussed in Section 3.4.7.

3.3.4 SRv6 Controller

In our architecture, a TCP client is able to specify arbitrary functions to apply to its traffic. However, keeping track of all the functions, parameters, and proxies addresses represents a significant amount of complexity. This complexity can be abstracted by a central SDN-like controller. We leverage the *SDN Resolver*, which is a DNS-based, SRv6 controller introduced in [Leb17, LJC+18]. Before establishing a connection, the client sends a request to the controller with the address of the server and a list of functions to apply to the traffic. The controller then computes a path that matches the request and returns an SRH to the client. A key element of this controller is that the SRH returned to the client does not contain the full list of segments. Instead, it contains only one segment, called the *binding segment*. The access router of the client is configured by the controller to translate this binding segment into the full list of segments. This abstraction enables the clients to be oblivious to changes in the SRH induced by, *e.g.*, a network failure. The architectural and implementation details of *SDN Resolver* are available in [Leb17, LJC+18]. Note that the DNS protocol serves as an example, that can be replaced by any ad-hoc application-facing protocol.

3.3.5 Security Considerations

The ability to execute and chain arbitrary functions in the network has obvious security implications. To restrict the privilege of using SRv6Pipes proxies, we can leverage the central controller presented in the previous section, as well as its *binding segment* mechanism. By configuring all access routers to accept only SRHs with known binding segments, we can effectively prevent an uncontrolled usage of network functions. The decision to accept or deny the use of a given set of functions is made by the controller, which can identify clients through independent channels [Leb17].



(a) Traversal of a SYN packet through the proxy. The SRH is recorded for the 5-tuple.

(b) Traversal of data packets.

(c) Traversal of return packets.

Figure 3.3: Overview of possible data paths within SRv6Pipes.

3.4 Implementation

To demonstrate the feasibility of our approach, we implemented a prototype of our solution by extending the implementation of IPv6 Segment Routing in the Linux kernel [LB17]. The main new component of our prototype is a transparent, SR-aware, TCP proxy. For this, we extended the kernel implementation of SRv6 with a new type of function. An overview of the various data paths in our prototype is shown in Figure 3.3.

To ensure that our proof of concept could easily be used to reproduce our results on any off-the-shelf hardware, we implemented it using the regular Linux mechanisms. Alternatives solutions are discussed in Section 3.8.

3.4.1 Transparent SR-Aware TCP Proxy

The core objective of our proxy is to process and relay TCP streams between two segments of a segment routed path. To achieve this, the proxy must (i) intercept and terminate incoming TCP flows, (ii) optionally apply transformation functions to the bytestreams, and (iii) initiate and maintain the corresponding TCP flows to the next segment of the path.

To intercept TCP flows, the proxy must accept connections towards pairs of IP/port that are not local to the machine, which is not possible by default.

The Linux kernel provides the `TPROXY` iptables extensions, enabling such interceptions. It works by redirecting all packets matching an iptables rule towards a local IP/port pair. The proxy is then able to intercept the corresponding TCP flows by listening to this local pair.

Once a TCP flow is intercepted and terminated, the proxy needs to retrieve the associated `SRH`, decrement its segment pointer, and install it on the corresponding outbound socket. The `IPV6_RECVRTHDR` socket option could be used to fetch any attached Routing Header (RH) as ancillary data, using the `recvmsg()` system call. However, this feature is only implemented for datagram protocols such as UDP, where a single RH is associated to each datagram. In bytestream protocols such as TCP, packets can be merged and the $1:1$ mapping to RHs is lost. In our prototype, we rely on the `SRH` included in the SYN packet of a given TCP flow. As the kernel does not expose Routing Headers for TCP flows, we leverage the `NFQUEUE` iptables extension to capture SYN packets in user space. The proxy opens a netlink channel with the kernel and receives through it all SYN packets matching the corresponding iptables rule. Then, the proxy extracts the 5-tuple and the `SRH` from the SYN packet and stores them in a `flows_SRH` map. Finally, the packet is reinjected into the kernel. Following its normal data path, the SYN packet will trigger a connection request to the proxy. Using the 5-tuple, the proxy is then able to retrieve the `SRH` previously stored in the `flows_SRH` map. While capturing every packet in user space can severely degrade the performances, our solution does not suffer from such degradation as we only capture the first packet of each flow.

After having intercepted a TCP flow and extracted its `SRH`, the proxy must establish the corresponding TCP flow to the next node of the path. To achieve this, the proxy creates the outbound socket and attaches the corresponding `SRH`. Additionally, the connection must appear as originating from the actual source of the flow. Using the `IP_FREEBIND` socket option, the proxy is able to bind to a non-local IP/port pair. Finally, the connection is established and data can be exchanged.

Once both connections (inbound and outbound) are established, the proxy only needs to forward data coming from one socket to the other one, after going through an optional transformation function. In our prototype, we use an application-level buffer to transfer data from one connection to the other. Another possible solution would be to use the `splice()` system call to let the kernel directly move data between file descriptors. However, this solution prevents the proxy from actually modifying the data. Our approach allows the implementation of arbitrary transformation functions. The termination of connections is straightforward. Once one socket is closed, any in-flight data is flushed and the other socket is also closed.

We implemented a multi-threaded architecture, enabling the proxy to

scale with the load. One dedicated thread handles the NFQUEUE channel, receives the SYN packets, and populates the flows_SRH map accordingly. A configurable number of threads (typically one per CPU thread) accept incoming connections, establish the outbound connection, and process the data exchanged between them. Each of these threads leverages the SO_REUSEPORT socket option, enabling them to simultaneously listen to the same local IP/-port. The result is that the kernel maintains distinct accept queues for each thread. Consequently, incoming connections are equally load-balanced across the running threads. To minimize the size of the stack space used in the function handling the data, each thread allocates its own copy buffer.

3.4.2 Kernel Extensions

When a packet to be processed by the proxy enters the kernel, its IPv6 destination address corresponds to the local proxy function. However, the TCP checksum was originally computed for the actual destination of the packet. As such, it is transiently incorrect, due to the SR-triggered change of destination address. Additionally, the packet will be associated to the proxy's local socket by the TPROXY module, and subsequently injected in the local stack. However, the segment pointer of the associated SRH is non-zero. The packet will thus enter the SRH processing and the kernel will attempt to forward it to the next segment, bypassing the local TCP processing [LB17].

To address those two issues, we extend the SRv6 kernel implementation available in Linux 4.14 and add a new type of function called End.VNF. This function takes one parameter (an egress interface) and performs the following actions. First, it updates the destination address of the packet to its final destination. Then, it sets the segment pointer to zero[1]. Finally, it injects the resulting packet into the specified egress interface using netif_rx(). In our prototype, we leverage a virtual dummy interface (nfv0). As a result, all packets to be intercepted by the proxy are received through this particular interface and are thus easily distinguished from background traffic.

3.4.3 System Configuration

To instantiate the proxy, a non-trivial configuration of iptables and routing tables is required. An example of this configuration is shown in Figure 3.4. The first two lines create the nfv0 interface to receive all packets to be intercepted by the proxy. Lines $3-5$ create a DIVERT iptable chain that sets the mark 1 on packets and accepts them. Line 6 creates an NFQUEUE rule that matches all SYN packets whose destination address corresponds to the

[1] As the SRH of the SYN packet was previously extracted by the proxy, this information is not lost.

```
1: ip link add nfv0 type dummy
2: ifconfig nfv0 up
3: ip6tables -t mangle -N DIVERT
4: ip6tables -t mangle -A DIVERT -j MARK --set-mark 1
5: ip6tables -t mangle -A DIVERT -j ACCEPT
6: ip6tables -t mangle -A PREROUTING -d \$PROXY_FUNC_ADDR -p tcp --syn
 -j NFQUEUE --queue-num 0
7: ip6tables -t mangle -A PREROUTING -i nfv0 -p tcp -j TPROXY
 --tproxy-mark 0x1/0x1 --on-port \$PROXY_LOCAL_PORT
8: ip6tables -t mangle -A PREROUTING -p tcp -m socket -j DIVERT
9: ip -6 rule add fwmark 1 table 100
10: ip -6 route add local ::/0 dev lo table 100
11: ip -6 route add \$PROXY_FUNC_ADDR/128 encap seg6local action End.VNF
 oif nfv0 dev eth0
12: sysctl net.ipv6.conf.nfv0.seg6_enabled=1
```

Figure 3.4: System configuration for the proxy.

local proxy (`PROXY_FUNC_ADDR`) and sends them to queue number 0. Line 7 matches all TCP packets received on interface `nfv0` and sends them to the `TPROXY` target. The latter sets the mark 1 on those packets and associates them to a socket bound on a local `PROXY_LOCAL_PORT` port. Line 8 matches all TCP packets that can be associated to an open socket and sends them to the previously configured `DIVERT` chain. In practice, this rule catches the inbound return packets that are not caught by the two previous rules. Line 9 creates a routing rule instructing the kernel to lookup table 100 for all packets having mark 1. Line 10 creates a single routing entry into table 100 that matches all packets and sends them in the local stack (instead of forwarding them). Line 11 creates an SRv6 routing entry that matches all packets towards `PROXY_FUNC_ADDR` and applies the `End.VNF` function, using `nfv0` as the egress interface[2]. Finally, line 12 enables the processing of SRv6 packets on interface `nfv0`.

3.4.4 Configuration optimization

To cope with high bandwidth links, several optimizations can be applied to the system's configuration. During our measurements, we fine-tuned some of these parameters to optimize the behavior of the proxy. While the exact value of these parameters depends on the system (CPU, amount of RAM, NIC model, etc), some of the parameters worth considering are:

- `net.core.rmem_*` : size of the socket receive buffer

[2]While this interface is considered egress from the point of view of `End.VNF`, packets are actually received on that interface and it is thus considered ingress for the next components in the datapath.

- `net.core.wmem_*` : size of the socket send buffer
- `net.core.optmem_max` : ancillary buffer size per socket
- `net.core.netdev_max_backlog` : maximum number of packets allowed to be queued on a particular interface
- `net.ipv4.tcp_rmem` : size of the TCP receive buffer
- `net.ipv4.tcp_wmem` : size of the TCP send buffer
- `net.ipv4.tcp_mem` : size of the memory for all TCP applications

As described in 3.4.1 we implemented a multi-threaded architecture for our proxy. Typically, one thread per CPU thread, but we recommend to experiment with different settings, based on the architecture. Another setting is the receive flow steering. By setting the IRQ affinity, it is possible to assign the TX/RX queues of the network card to a specific CPU core. In this configuration, we recommend to assign each interface's TX/RX queue to a different CPU, which is the default behavior when using Receive Side Scaling (RSS) [RSS].

3.4.5 Modular Transformation Functions

To support transformation functions in a modular way, our SRv6Pipes proxy leverages Linux dynamic libraries. Functions can be compiled in `.so` (shared object) files. Those files are independent modules that can be loaded and unloaded at run-time by the proxy. Each module exports an `all_funcs` symbol. This symbol refers to an array of `func_desc` structures. Each of those structures describes a single transformation function, through the following symbols. The `func_init()` symbol is called once, on module load. It registers the function with a given function identifier, which is passed in the IPv6 destination addresses (see Section 3.3.3). The `func_spawn()` symbol is called each time a new intercepted TCP flow matches the function identifier. Any parameter passed in the low-order bits of the IPv6 destination address is passed as argument. The role of this symbol is to initialize per-connection data. The `func_process()` symbol is the actual transformation function. It reads data from an input buffer and writes the transformed data in an output buffer. The `func_despawn()` symbol is called at connection termination and it frees previously allocated per-connection data. Finally, the `func_deinit()` symbol is called at module unload and de-registers function identifiers.

Such an architecture enables to easily add, modify, and remove transformation functions, without updating or restarting the proxy's binary.

3.4.6 Limitations of Transparent Proxies

During our tests, we initially measured poor results such as requests timing out and unexplainable high latency for some requests. These figures appeared only for benchmarks with multiple concurrent connections. After some troubleshooting, the cause was determined as follows: lines 7 and 8 in Figure 3.4 were initially swapped. As a result all packets received on `nfv0` and associated to an open socket would match the `DIVERT` rule and be directly injected in the local stack without processing by `TPROXY`. This would not be an issue when either no open socket is found (new connection), or when the open socket actually matches packets (regular data transfer). However, when many connections are created, the same source port may be reused from a previous connection. As the source and destination addresses and port are the same, the connection reusing the source port will have the same 5-tuple. While the previous connection may be terminated on the client side, the proxy's kernel still maintains a `TIME_WAIT` state, to deliver potentially lost acknowledgments. When the client sends a SYN packet with the same 5-tuple as an old connection in a `TIME_WAIT` state, the kernel wrongly associates it with the stale socket and the `DIVERT` rule matches. Usually, when the Linux kernel receives a SYN packet matching a `TIME_WAIT` socket while there is an active listener on the destination IP/port, it destroys the stale socket and properly redirects the SYN to the listener. However, in our case, there is no matching active listener. Indeed, the proxy listens to a different IP/port, and it is the role of `TPROXY` to redirect the packets to the proxy. As a result, the kernel replies with an acknowledgment that does not correspond to the new connection. The client rejects it and replies with a RST, prompting the proxy's kernel to destroy the stale `TIME_WAIT` socket. However, from the client perspective, no valid response was received for the SYN packet and it must be retransmitted. This accounts for the high latency and timeouts measured for some of the requests. The second SYN packet will always succeed, as the stale socket was destroyed by the RST[3]. The `TPROXY` module includes a workaround that immediately destroys any `TIME_WAIT` socket and correctly redirects the SYN packet to the listener. We fixed this issue by applying the `TPROXY` rule before the `DIVERT` rule, as shown in Figure 3.4.

It might happen that legitimate traffic with identical 5-tuples is sent to the proxy's kernel for regular forwarding instead of local processing. For instance, a client establishing numerous short-lived connections might end up reusing the same source port, resulting in the same 5-tuple. If this traffic is not distinguished from inbound return traffic, this might result in transient traffic blackholing. The fact that, we discriminate between background traffic and inbound return traffic and send the inbound return traffic through the

[3]This behavior is configurable through the sysctl `net.ipv4.tcp_rfc1337`.

nfv0 interface prevents this problem.

3.4.7 Return Traffic

The previous sections detailed the processing of the upstream traffic (from client to server). However, if the middleboxes are not located on-path, the downstream traffic (from the server to client) must also be augmented with an SRH. This is also necessary to enable asymmetrical processing functions, *i.e.*, using different transformation functions depending on the direction of the traffic. To achieve this, different options exist.

The straightforward option is to simply "*reverse*" the SRH received from the client or from the previous proxy. Each proxy can simply apply the segments of the initial SRH in reverse order. While this solution is simple and does not incur a significant overhead, it as a major limitation: the segments must necessarily be symmetrical, making asymmetrical processing functions impossible.

To enable asymmetrical processing functions, another option is to embed the return SRH in a TLV extension of the initial SRH. With this solution, after inserting the SRH, the client inserts a TLV to the socket before establishing the connection. Then, each proxy and the server extract the SRH to be used on the return path from the TLV received in the initial packet (SYN). The TLV could also be transmitted with every upstream packet, but this would increase the overhead. With this TLV, it is important to note that the return path must include every proxy that is present in the upstream path, but that others segments, e.g. corresponding to specific paths or routers, can be added or suppressed.

In our prototype, we implemented the second solution by modifying the Linux kernel to add support for such a TLV. When a new TCP socket is created after receiving an SR-enabled SYN packet containing the return-path TLV, this return path is extracted and installed as an outbound SRH for the newly created socket. If the proxies are located in-path, our prototype can also work without an SRH on the return path. This is realized using the DIVERT rules shown in Figure 3.4. In Section 3.5, we evaluate this on-path mode.

3.5 Evaluation

In this section, we use microbenchmarks to evaluate the performance of our prototype in a lab. For this evaluation, we use three Linux PCs connected with 10Gbps interfaces as shown in Figure 3.5.

The client is a 2,53Ghz Intel Xeon X3440 with 16 GB of RAM. M1 and the server use the same hardware configuration but with only 8 GB of RAM. They are all equipped with Intel 82599 10 Gbps Ethernet adapters and use 9000 bytes

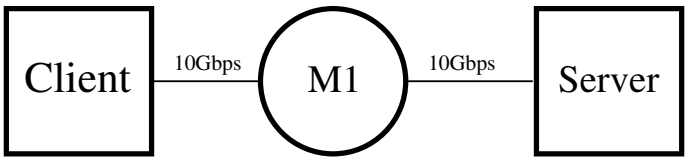


Figure 3.5: Lab setup. M1 can be configured as router or proxy.

MTU. They all use our modified version of the latest IPv6 Segment Routing kernel based on the Linux kernel version 4.14. The server runs `lighttpd` version 1.4.35. The client uses `wrk` [wrk] 4.0.2-5 to load the server with HTTP 1.1 requests. We slightly modified `wrk` to add an IPv6 `SRH` as a socket option when creating TCP connections. M1 can be configured either as a router or with our transparent proxy. When used as a router, we configure static routes and use the standard Linux IPv6 forwarding.

3.5.1 Maximum throughput

First, we compare the performance of one of our proxies against the performance of a Linux router running on the same platform. In this setup, our client uses `wrk` [wrk] to simulate 200 web client downloading static web pages of given sizes during 120 seconds. It uses 8 threads with 25 connections per thread. The proxy was configured with a virtual function that directly copies that bytestream without any processing.

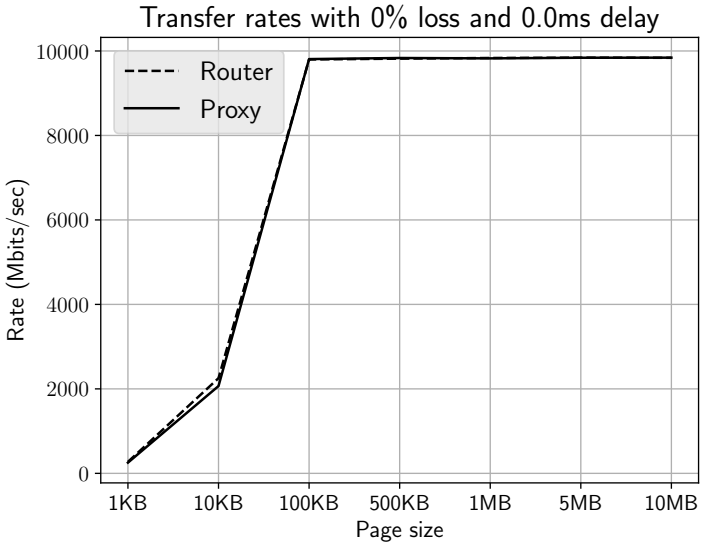


Figure 3.6: Raw throughput.

Figure 3.6 shows the total transfer rate when the client is downloading web pages. This figure shows that there is no significant difference in transfer rates between our proxy and the router. With 10 MB files, our proxy reaches a throughput of 9841 Mb/s where the router achieves 9838 Mb/s. A closer look at the small page sizes in Figure 3.6, shows that our proxy slightly underperforms the router. With 1KB files, our proxy achieves a rate of 253 Mb/s, while the router achieves a rate of 272 Mb/s. This is confirmed by Figure 3.7 that shows the number of requests per second.

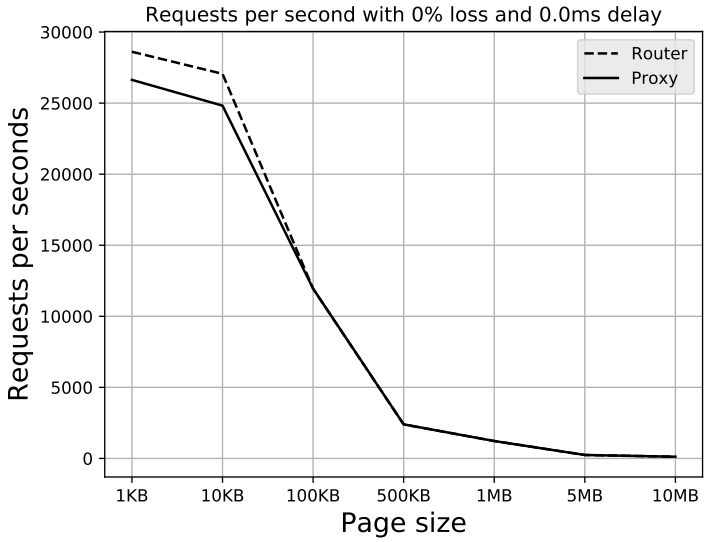


Figure 3.7: Number of requests per second in a simple setup.

In term of requests per second, for 1 KB files, our proxy completes 26634 requests per second, while the router completes 28613 requests per second. This difference in performance between large and small files can be explained by the fact that when our proxy receives a new connection from the client, it needs to establish a new connection to the server before starting to forward packets. With smaller files, there are significantly more three-way handshakes to perform, making this overhead more important while this cost is amortized for larger files. With 100 KB files, the number of requests per second is already on par at ≈11945 requests per second for both the proxy and the router.

3.5.2 Stability of the performances

To examine the stability of the proxy's performances, we ran the experiments several times to compare the results of different runs. This confirmed that our

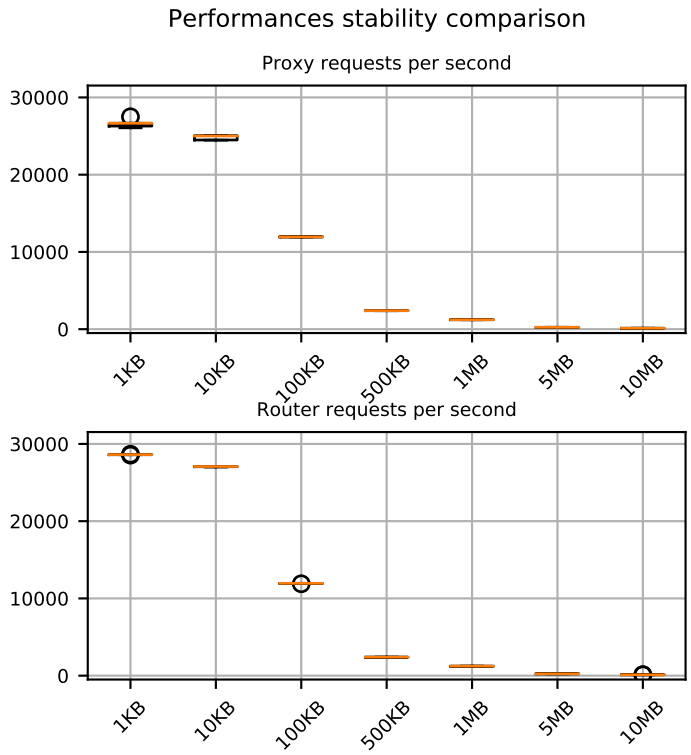


Figure 3.8: Performances stability comparison between the proxy and the router in term of requests per second.

results were as stable as the results with the router. The only significant deviation observed is the proxy being ≈3% faster for the first 1KB run compared to all other runs for 1KB files. This difference only applies to this specific run. Figure 3.8 shows the stability of the performances of our proxy and compares it to the router. As discussed previously, this figure clearly shows that for 1KB files, our proxy slighlty underperforms the router. Overall, in terms of stability, our results are comparabale to the router results, with a notable exception for 1KB file sizes, where we observe an outlier that represents a ≈3% deviation compared to the mean of the measurements.

3.5.3 Impact of packet losses and latency on the proxies

The previous section explored the maximum rate that our proxies can sustain. In those measurements, the TCP stack running on M1 did not have to buffer packets or handle retransmissions. As those operations can affect its performance, we added `netem` to simulate different delays and different packet loss ratios.

We start by adding a 1% loss and a 25ms delay on the four links of Figure 3.5. This corresponds to an end-to-end loss of ≈4%, and an end-to-end latency of 100 ms. The results of this measurement are shown on Figure 3.9. Under such circumstances, our proxy outperforms the router. This is not surprising since in this setup, our proxy acts as a Performance Enhancing Proxy (PEP). While Figure 3.9 clearly shows a large improvement for large file sizes, our measurements indicated that this is also true for small file sizes. This can be explained by the fact that when M1 is configured as a router all packet losses need to be recovered end-to-end. When a packet is lost on the same link with our proxy, the retransmission is done by the proxy. The minimum Round Trip Time (RTT) between the client and the server being 100 ms while the minimum RTT between the client and the proxy is 50 ms, it is faster to retransmit from the proxy. Our proxy brings the data "closer" to the client, and minimizes the impact of losses.

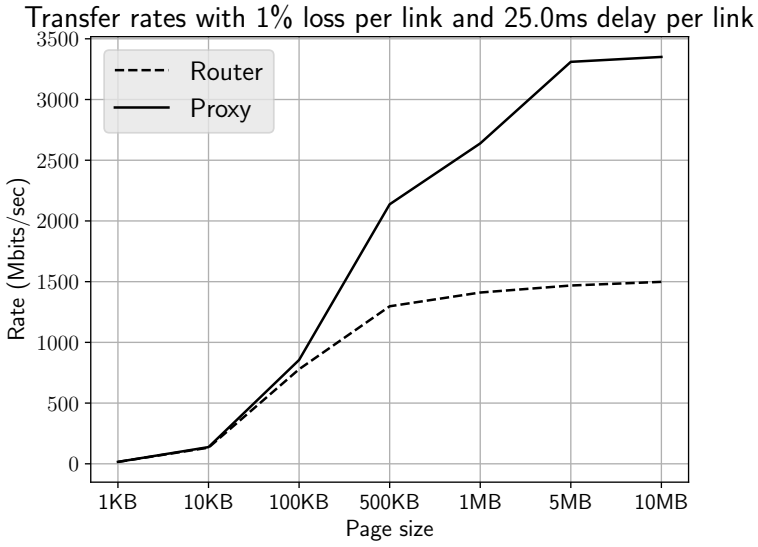


Figure 3.9: Transfer rate with 1% of loss and 25ms of latency per link.

To confirm our findings, we run the same measurement, but adding latency and loss only on the link between the server and the proxy, the objective being to mimic a network where the loss would happen only on the link between the proxy and the server. To replicate our previous configuration, we add 2% of loss per link, to get an end-to-end loss of ≈4%, and 50ms of latency per link to get an end-to-end latency of 100ms. As shown in Figure 3.10, under such conditions, the proxy and the router are both significantly affected by the performance degradation in the same fashion, confirming our findings.

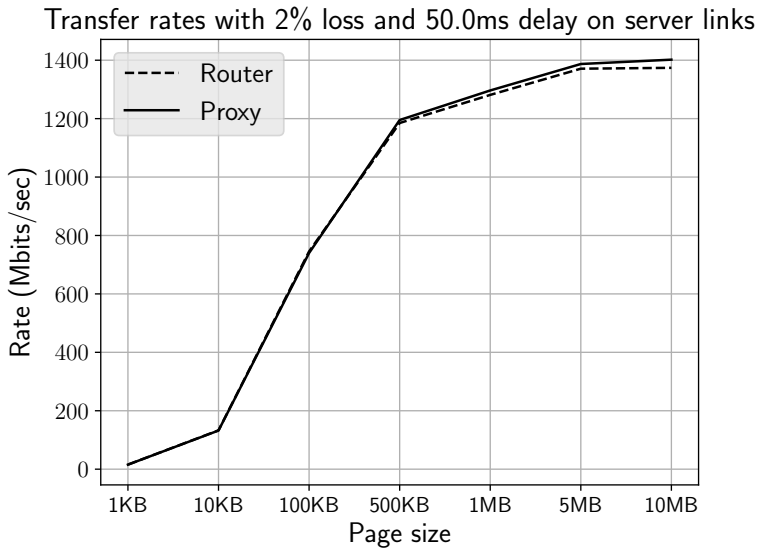


Figure 3.10: Transfer rate with 2% of loss per link and 50ms of latency per link between the proxy and the server.

3.5.4 CPU-intensive Virtual Functions

With our architecture, various types of Virtual Functions can be implemented. Some like a PEP simply proxy the connections and do not need to process the payload. Others like DPIs, transparent compression or transparent encryption need to process the payload and thus consume CPU cycles. To measure the impact of the Virtual Function on the performance of our proxy, we developed a simple microbenchmark that performs $2 \times n$ passes over the bytestream and XORs each byte with a key at each pass. This VF leaves the bytestream unmodified, but consumes both CPU and accesses memory.

The results with this microbenchmark are shown in Figure 3.11. When our VF performs two passes on the bytestream, the maximum throughput is similar to the one we obtained without bytestream modification in Figure 3.6. When the VF performs four passes on the
bytestream, the maximum throughput with pages larger than 100KB is divided by 2. This throughput continues to drop with the CPU load on the VF. To confirm that the reduction in throughput was due to the CPU intensive computations, we ran `perf` [perf] that yielded 96% of cycles spent in the XOR function.

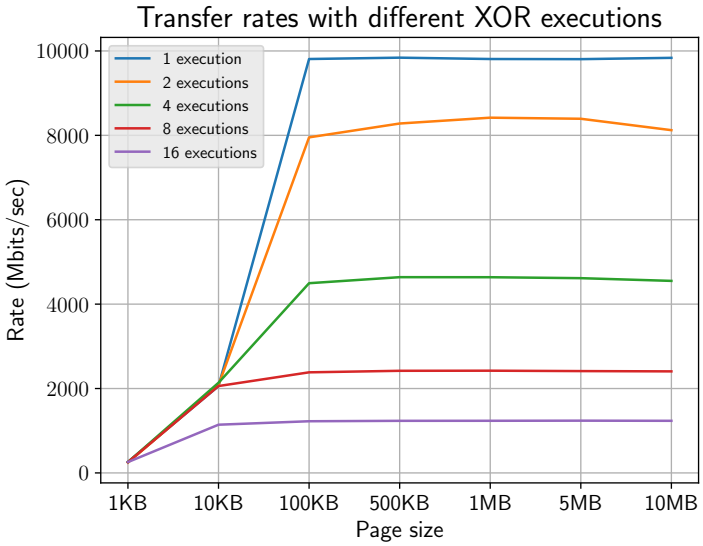


Figure 3.11: Maximum throughput with Virtual Functions performing n passes over the bytestream.

3.5.5 Chaining middleboxes

In this section, we chain two proxies to demonstrate the feasibility of our architecture. Figure 3.12 describes the configuration of the lab that we used for this evaluation.

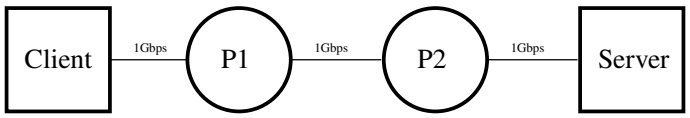


Figure 3.12: Middleboxes chaining evaluation setup with P1 and P2 acting as proxy.

With our architecture, middleboxes can be used in chains where one middlebox performs the opposite function of the previous one. Typical examples include transparent compression/decompression or transparent encryption. To demonstrate this use case, we implemented a VF that simply XORs each byte of the bytestream with a constant.

In this configuration, the client is connected to the server through two middleboxes that will be used as proxies or routers. Due the limitations of our lab, these measurements had to be run with 1 Gbps links instead of the 10 Gbps links as we only have 3 servers with 10 Gbps network cards. While this is unfortunate, we argue that 10 Gbps links were essential to measure the

overhead of the proxy under heavy load, but are less important to demonstrate the feasibility of middleboxes chaining. In this configuration, the client is a 2 Ghz AMD Opteron 6128 with 16 GB of RAM, P1 is using the server used by the client in the previous sections, P2 and the server are using the same machines. When two such middleboxes are used in sequence, the bytestream output of the downstream one is the same of the input of the upstream one. This is illustrated in Figure 3.13.

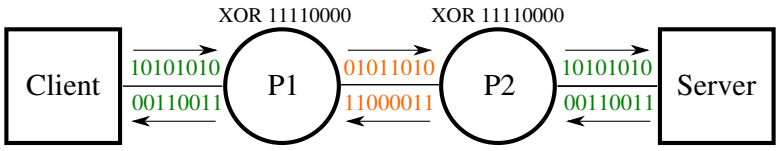


Figure 3.13: Demonstration of middlebox chaining with simple XOR transformations.

Figure 3.14 shows that with the two chained middleboxes, the maximum throughput was the same as when passing through two routers. This is expected given the results of Figure 3.11 with 10Gbps interfaces.

Figure 3.14: Transfer rate of wrk with 2 proxies applying a XOR.

3.5.6 Commodity hardware

One of the advantage of our solution is that it is deployable on off-the-shelf hardware. We demonstrate this by evaluating the performances on a home

router running an open-source firmware.

OpenWRT

OpenWRT [openwrt] is the most frequently used open source embedded operating system based on Linux. It supports 50 different platforms and about 3500 optional software packages. We modified the latest snapshot (r7846) to include our kernel and iproute patches. Then, we modified our proxy to integrate it into a standard OpenWRT package [howtopackage], that can easily be installed on any device running OpenWRT.

Turris Omnia

Developed by cz.nic[cznic], the Turris Omnia [turris] is an open-source router targeting the small office/home office market. It uses a 1.6 GHz dual-core Marvell Armada 385 ARM CPU with 1GB (extensible to 2GB) of RAM and features 5 GBit LAN ports and 1 WAN port. While the Turris Omnia normaly runs TurrisOS, a fork of OpenWRT, it supports the original OpenWRT. Given its modern hardware and the fact that some of the use cases described in 3.2 also target the small office/home office market, we think that using this router is a realistic option to run our evaluations on.

As shown in Figure 3.15, to evaluate our proxy, we use a Turris Omnia with 1 GB of RAM and 2 Linux PCs.

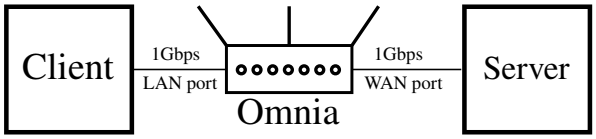


Figure 3.15: Turris Omnia setup. The Turris Omnia can be used as router or proxy.

We use our modified version of `wrk` again to simulate 200 web clients downloading static web pages of given size during 30 seconds. In this setup, `wrk` uses 8 threads with 25 connections per thread. On the Turris Omnia, the proxy uses 4 threads and does not apply any modification to the payload.

Figure 3.16 shows the total transfer rate reported by the client when downloading web pages. This shows that for files between 100 KB and 10 MB, there is no significant difference in terms of throughput between our proxy and the router. For 10 MB files, both the proxy and the router achieves a throughput of 988 Mb/s. For smaller files however (1 KB and 10 KB), we observe that the proxy achieves a lower throughput of 194 Mb/s while the router achieves 237 Mb/s.

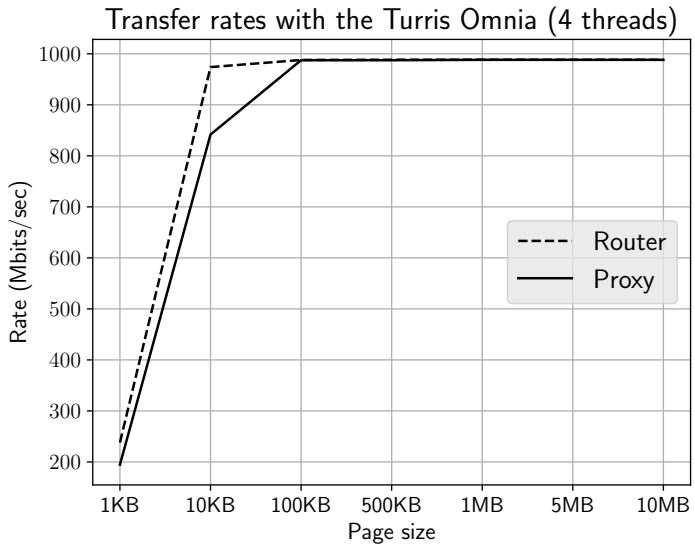


Figure 3.16: Throughput reported by the client when using the Turris Omnia.

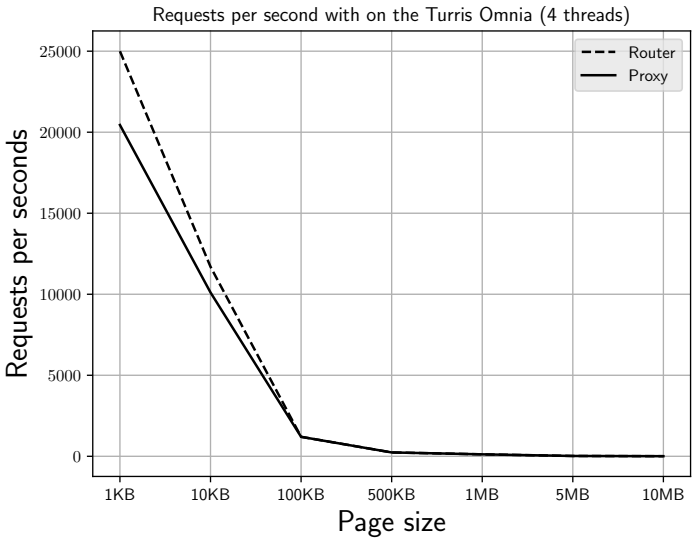


Figure 3.17: Number of requests per second with the Turris Omnia.

This difference is more visible on Figure 3.17, showing the number of requests per second during the same benchmark. We already observed such a difference in Section 3.5.1, on a smaller scale. In that section, we explained the difference by the fact that there are significantly more three-way-handshakes

for smaller file sizes, making the overhead more important while this cost is amortized for larger files. Given the fact that the three-way-handsakes are more CPU intensive than forwarding data when the connection is established, the difference is more important here because the CPU is less powerful than the Xeon X3440 we used before.

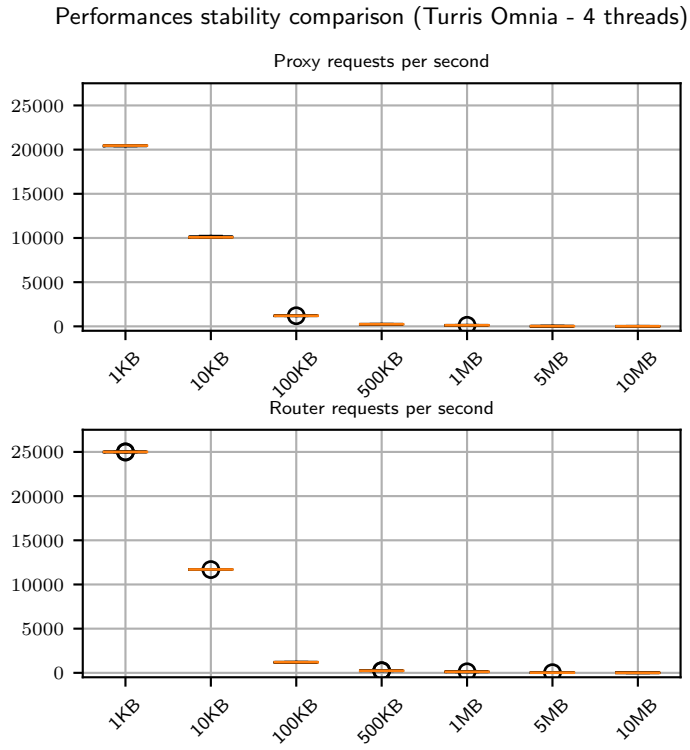


Figure 3.18: Performance stability comparison between the proxy (with 4 threads) and the router in term of requests per second.

We compared the variability of the results over multiple executions of the same benchmark on both the proxy and the router. Figure 3.18 shows that there are no significant differences between the router and the proxy in terms of variability.

In Section 3.4.4 we highlighted the importance of configuring the number of threads used by the proxy to accept connections. Figure 3.19 demonstrates this by showing the performance of our proxy running on the Turris Omnia while using only one thread to accept connections (another thread is always used for the `NFQUEUE`).

The Marvell Armada 385 being a dual-core CPU, one could expect that using one core to accept connections while the other one is used for the `NFQUEUE`

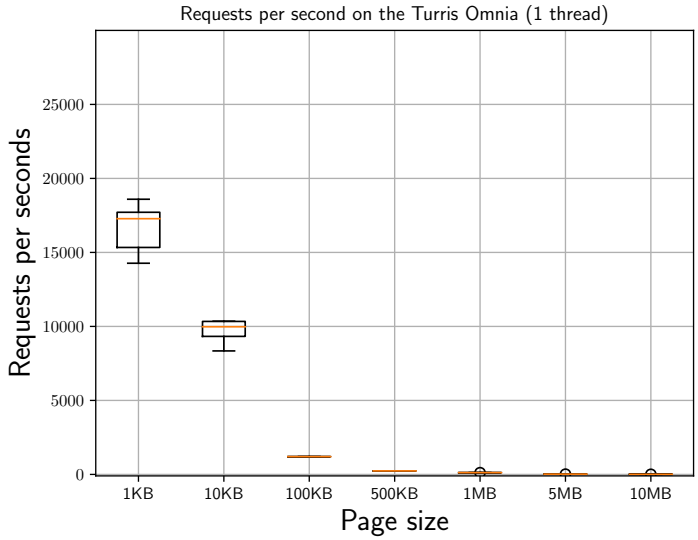


Figure 3.19: Requests per second with the Turris Omnia using 1 thread to accept connections.

thread would be the best setting. However, as shown by Figure 3.19, with this configuration, we see an important drop in performances for 1 KB files and an increased variability of the results. During our experiments, we found that 4 threads to accept connections is a good setting on the Turris Omnia, resulting in 2 threads per-CPU core. For the Xeon X3440 machines used previously, we found that 1 thread per-CPU thread was a good setting, however this CPU uses hyper-threading, resulting in roughly 2 threads per-CPU core.

3.6 Related Work

AbdelSalam et al. propose in [ACF+17] to use IPv6 Segment Routing to support Virtual Network Function Chaining and implement a prototype as a Linux kernel module. They leverage namespaces to support virtual network functions but only support packet-based functions while our solution leverages the Linux TCP stack to provide a bytestream abstraction to the network functions. In FlowOS, Bezahaf et al. [BAM13] proposed a Linux kernel module that exposes a bytestream abstraction to network functions but they do not describe how flows are routed through the network functions. NetVM [HRW14] leverages virtualization techniques and a user-space packet processing platform to provide fast, chainable network functions in virtual machines. Their work focuses on packet processing and does not consider bytestream functions. Other solutions such as XOMB [ABK+12] focus on

the system aspects of implementing virtual functions to support middleboxes through a flexible programming model. Our architecture leverages IPv6 Segment Routing to forward the packets to the middleboxes. Another related work is `/dev/stdpkt` proposed by Utsumi et al in [UTE17]. `/dev/stdpkt` uses the Linux Kernel Library to implement virtual functions that can be chained together.

3.7 Conclusion

Given its ability to enforce precise network paths for specific flows, IPv6 Segment Routing appears to be an excellent candidate to support middleboxes in entreprise networks. We leverage this IPv6 extension in our architecture designed for enterprise networks. Its main benefit is that the middleboxes are explicitly exposed. This significantly improves the manageability of the network. Our architecture supports both middleboxes that operate on a per-packet basis (e.g. NAT, stateless firewalls) and those that need to process bytestreams (e.g. DPI, Application Level Gateways, ...). For the latter, we use transparent TCP proxies that process the IPv6 Segment Routing Header. We implement this architecture in the Linux kernel and evaluate its performance with various benchmarks in our lab. Our measurements indicate that our architecture is well suited to support middleboxes that process bytestreams.

3.8 Future Work

With SRv6Pipes, we implemented a proof of concept using the regular Linux mechanisms. While kernel bypass techniques such as DPDK or user-space TCP stacks like mTCP allow significant performance boosts, they are often specific to a subset of network hardware. By leveraging the kernel data-path, our solution remains generic and can be deployed on any Linux-supported hardware, ranging from high-end servers to home routers. Should an operator require performance only available through kernel bypass techniques, our high-level network architecture would remain identical and our userspace implementation of the proxy would require minimal changes to plug-in with a DPDK-like library. These modifications can be realized as future work.

Chapter 4

Steering transport flows in Multipath networks

In this chapter, we propose two solutions to efficiently leverage the path diversity in datacenter and enterprise networks. First, we present FlowBender, a flow-level adaptive routing technique for improved latency and throughput. Then, we propose a solution that goes further than FlowBender and leverages IPv6 Segment Routing and eBPF to efficiently steer flows.

4.1 FlowBender: re-routing flows using Equal Cost Multipath

Datacenter networks provide a high path diversity for flows between any pair of hosts. Typical multi-stages Datacenters topologies like fat-tree [AFLV08, Lei85] provide a large number of paths between hosts. This is illustrated in Figure 4.1. Large-scale operators rely on network performances to improve user experience. Because user-facing responses are constructed by aggregating the results from several servers, the tail latency of the individual flows directly affects response times and quality. In this context, efficiently using these multiple paths becomes a critical requirement to guarantee that the flows fully take advantage of the entire network infrastructure.

At this level of requirements and scale, Datacenters needs a load-balancing technique that: (1) spreads the traffic evenly (2) scales easily (3) does not impact the performances negatively (i.e by creating too much reordering). Presented in Chapter 1.3, `Equal Cost Multi-path` (ECMP) is the standard solution to load-balance flows in Datacenters. By hashing the packet headers to select a path, it allows to spread the flows evenly in terms of number of flows per-link. Furthermore, ECMP is stateless. If the header fields are correctly chosen (i.e the 5-tuple of a connection) a flow always uses the

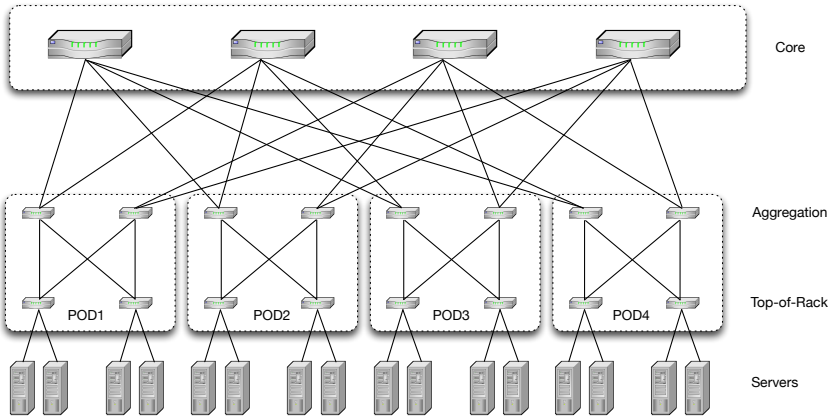


Figure 4.1: Fat-tree Datacenter Network Topology

same path, thus limiting reordering. While being the reference, `Equal Cost Multi-path` still suffers from shortcomings. This solution works well when all the flows share the same characteristics in terms of duration and size, but might struggle when the flows are too heterogeneous. Benson et al. [BAM10] demonstrated that, in Datacenters, a few long flows contribute to an important fraction of all the traffic. This case is sub-optimal because ECMP is "blind" to the type of flow, meaning that this solution cannot differentiate a long flow from a short flow. In this situation, several long flows can collide on a path, creating long-lasting congestion, while other paths could be underutilized. Because it has no way of sensing congestion just by hashing the packet headers, ECMP can not react to this kind of collision. The same applies when a path breaks, ECMP is not be able to re-route the traffic and continues to forward the packets into a black-hole. While solutions like `De-Tail` [ZDM+12] and `RPS` [ADK13] haven been proposed to address ECMP's limitations, they incur a high degree of re-ordering and `De-Tail` requires hardware changes to the switches. In this scenario, Multipath TCP could be considered, however its deployment requires important modifications to the end-host TCP/IP stack.

To allow an efficient load balancing while reacting to congestion, we propose FlowBender, an end-host-driven load balancing scheme that is dynamic and incurs little to no packet re-ordering. FlowBender offers both simplicity and high performance. Its main idea is to address ECMP's shortcomings by re-routing a flow when its path is either broken or congested. FlowBender is a host based solution, that removes the need to make hardware changes and uses existing ECMP-networks. The fact that FlowBender only re-routes a flow when it detects that its path is broken or congested drastically reduces

out-of-order packet delivery.

In this Section, we first describe FlowBender's architecture. We then evaluate it with a real implementation and show that FlowBender reduces the tail latency over ECMP by more than 40% on average for large flows showing that FlowBender makes a strong case for a non-intrusive, end-host- and flow-level load balancing using the classic end-to-end principle [SRC84].

4.1.1 The architecture of FlowBender

We now explain the details of FlowBender, beginning with the insight that lead to its design.

Flow versus Packet-level Load Balancing

FlowBender operates in between the extreme of statically sending a flow on one path (ECMP) and that of spreading its packets across multiple paths simultaneously (e.g. RPS, DeTail, and MPTCP). Effcient load balancing does not necessarily mean that we have to simultaneously spread a flow across several paths and reassemble it at the receiver, especially when there is no congestion to start with. Instead, we target the simpler mechanism of shifting (rerouting) the entire flow to a different path only once it is congested or disconnected, thus avoiding sustained out-of-order packet delivery and any hardware updates to existing datacenter infrastructures.

Flow Control: Link versus Transport Layer

Link and transport-level congestion signals are important for realizing better rate control mechanisms, but they could be also leveraged for guiding our load balancing decisions as will be explained next. The question we are trying to answer now is: *which congestion signals should we leverage?* Link-level notification mechanisms such as PFC have faster reaction times compared to RTT-based ones such as Explicit Congestion Notification (ECN), especially when the congestion point is close to the traffic source itself. When the congestion point is far away from the traffic source, however, such link-level schemes can notoriously result in congestion spreading trees [Dal92, STJ03, DK89]. Rather than worrying about this phenomenon and adding complexity into our switches for load balancing based on PFC signals, our argument is that something simple like ECN has been already demonstrated to be prompt enough in propagating congestion information back to the sources. Of course, relying on end-to-end ECN signals means that we are targeting those longer flows that take several roundtrips to finish, which happen to be those flows carrying most of the network traffic anyway [BAM10]. Otherwise, in the hypothetical scenario where most of the traffic is generated by very small flows

only, ECMP should perform quite well handling such traffic given the much higher hashing entropy involved. Another important reason supporting our transport-layer choice is that, unlike link-level ones, protocols such as TCP can quickly detect link failures end-to-end, which can influence FlowBender to promptly avoid broken paths. With the clear motivation to pursue flow-level and RTT-based load balancing only once congestion occurs, we now proceed to discuss the details of FlowBender.

FlowBender

By design FlowBender's architecture is meant to be ultra-simple and compatible with today's commodity Datacenter hardware. It follows an approach that does not need additional hardware complexity or incur a high overhead due to complex multi-path techniques. In a nutshell, FlowBender *transmits a flow on a single path, through an ECMP-based network, and reroutes that individual flow only once it is congested.*

FlowBender has two main components: (1) a mechanism to detect path congestion and link failures at the host, and (2) a means for the end host to re-route a specific flow.

Detecting Congestion To detect congestion and link failure, we use standard TCP Timeouts (RTO) and Explicit Congestion Notification (ECN), a TCP /IP extension that is commonly used in today's Datacenters. ECN is a standard congestion notification scheme in today's Datacenters where *a congested switch marks every packet exceeding a desired queue size threshold, and the TCP sender keeps track of the fraction of ECN-marked ACKs every RTT.* If this fraction is larger than a certain threshold for any flow, it means that this flow is congested and should be rerouted. By monitoring RTOs and the fraction of packets marked by ECN, a host can decide whether to re-route a flow or not. Because it relies on end-to-end mechanisms, FlowBender is able to detect congestion and failures across the entire path of the flow. It is important to note that our approach is not strictly tied to one way of triggering and handling this feedback vs another.

Rerouting flows When a congestion or a failure is detected, we leverage ECMP to re-route the flow. As explained in Chapter 1.3, ECMP relies on a selection of fields, typically the 5-tuple, in the packet header to compute a hash. However, some already existing commercial platforms [BCMHash, cisco] allow to configure the hashing engine to hash upon other fields of the headers (e.g the VLAN ID field from the Ethernet header) or on programmable header offsets such as the Time-To-Live (TTL) field of a packet. This simple change in the configuration can be done without replacing the platform. We leverage

this feature and configure the hash function in the switches to compute the hash based on an additional "flexible" field such as the TTL in addition to the other fields. Per FlowBender, each TCP socket independently keeps track of the value `V` it should consistently insert into such a "flexible" hashing field. When a flow needs to be re-routed, we modify the value of `V` for this specific flow. Consequently, the value of the computed ECMP hash changes, and the flow is re-routed. Conceptually, this "flexible" field acts as a path ID for the flow, and changing it causes the flow to change path.

To summarize, each TCP socket sender (i.e. flow) independently keeps track of its value `V` and the per-RTT fraction of marked ACKs `F`. Once `F` exceeds a set threshold `T`, the current path is considered to be potentially congested and the packets of the corresponding flow are rerouted by changing the value `V` of the flexible field. The pseudocode for FlowBender's basic algorithm is shown in Figure 4.2. As is clear from its pseudo code and description, FlowBender's design is simple enough that its complete implementation requires only about 50 lines of kernel code on the hosts, and 5 lines of configuration code on the switches. Such simplicity is in stark contrast to the software complexity of schemes like MPTCP [PB+] and the hardware complexity of DeTail [ZDM+12].

---

for every RTT **do**
  $F \leftarrow num\_marked\_pkts/total\_pkts$
  if $F > T$ **then**
    $Change\ V$
  end if
end for

Figure 4.2: Basic FlowBender pseudocode

Another important feature of FlowBender that needs to be emphasized is its ability to recover from path failures. A packet could be at an advanced stage in its route where the only way for reaching the destination happens to be broken. In that situation, DeTail or PacketScatter would be unable to reroute around the broken path until the routing tables are updated. Because FlowBender also monitors the TCP Timeouts, a rerouting can be triggered once an RTO takes place, bringing the failure recovery time several orders of magnitude smaller.

Optimizing FlowBender Due to its simple design illustrated by Figure 4.2, FlowBender can easily be tuned to accommodate different requirements. For instance, to minimize the re-ordering, FlowBender could easily be readjusted

to reroute only after a flow is consistently congested (`T` is exceeded) for `N` consecutive RTTs. This could be done by adding a handful of lines (a variable to count the number of consecutive congested RTTs) to Figure 4.2. This is illustrated in Figure 4.3.

```
for every RTT do
  F ← num_marked_pkts/total_pkts
  if F > T then
    num_congested_rtts + +
    if num_congested_rtts >= N then
      num_congested_rtts ← 0
      Change V
    end if
  else
    num_congested_rtts ← 0
  end if
end for
```

Figure 4.3: Optimized FlowBender pseudocode

4.1.2 Evaluation

In this section, we present some evaluations of FlowBender. Additional evaluations may be found in [KVHD14].

Functionality Verification (Simulation)

To validate FlowBender's functionality, we use ns-3 [ns3] simulations. We simulate a fat-tree network [Lei85] analogous to the one depicted in Figure 4.1. The network has 128 servers, organized into four pods, each having four Top of the Rack (ToR) and four aggregation switches, with eight core switches interconnecting the pods (overall oversubscription factor of four from servers to core switches). Because 10 Gbps Ethernet is typical in today's datacenters, we use 10 Gbps point-to-point Ethernet links across our entire network. We configure the host delay to be 20 μs and the switch delay to be 1 μs, and we obtain a baremetal RTT of $2 \times 5 \times 1 + 4 \times 20 = 90\mu$s between two servers on different pods, which is realistic per today's datacenters RTTs. We set the parameters of DCTCP [AGM$^+$10] to match those in [AGM$^+$10]: (1) g, the factor for exponential weighted averaging, is set to $\frac{1}{16}$; and (2) K, the buffer occupancy threshold for setting the CE-bit, is set to 90 KB (typical for 10 Gbps links). So, our base case (ECMP) is DCTCP running over a

Flows	ECMP (ms)		FlowBender (ms)	
	Mean	Max	Mean	Max
8	588	1950	294	367
16	1468	5220	580	740
24	2515	9238	897	1144

Table 4.1: FlowBender's flow completion times relative to ECMP's

commodity datacenter network with ECMP-enabled switches. For FlowBender, we set T, the congestion threshold, to 5%, and N, the number of RTTs a sender must be congested before switching paths, to 1. Both DCTCP and FlowBender use an RTO_{min} of $10ms$. We start by evaluating FlowBender's efficiency in load balancing large flows to validate its functionality. In this experiment, we simultaneously initiate a small number of 250 MB flows from hosts on one ToR in a specific pod transmitting to hosts on another specific ToR in a different pod. We compare the average and worst completion times of the flows with FlowBender to that under ECMP. Because all flows are of equal size, as load balancing improves, both the mean and the maximum flow completion times improve. Furthermore, with better load balancing, we expect a tighter distribution of flow completion times i.e., the mean and the tail are close. Therefore, we can think of the ratio between the mean and the tail as a quantitative measure of the quality of load balancing.

In this experiment, we vary the number of flows as 8, 16 and 24 flows, which translates to an average number of 1, 2 and 3 flows per route respectively. Consequently, we expect the best flow completion times to be roughly 200, 400, and 600 milliseconds respectively (modulo the round-trip time delay and assuming instantaneous rate convergence to the fair share with no slow-start delays). Table 1 shows the mean and maximum flow completion times. As expected, we see that FlowBender improves ECMP's mean by 2x and maximum flow completion times by 5-8x respectively. Also, the ratio of the maximum flow completion time to the mean flow completion time is more than 3.3 with ECMP, while with FlowBender the ratio reduces to less than 1.3 implying a tighter latency distribution with lower variance.

Testbed Implementation

Our real implementation (testbed) has 15 ToR switches with 12 to 16 servers each. The servers are connected to the ToRs via 10 Gbps links, and the ToRs are interconnected via 4 aggregation switches with one 10 Gbps link to each of the 4 switches. In other words, each server has 4 distinct paths to reach any other server on the other ToR. Servers are running Linux 3.0, including the aforementioned FlowBender changes (less than 50 lines of code added

to the kernel) and the DCTCP implementation, and have their RTO set to 10 msec. We use standard ECMP-capable switches with a shared buffer space of 2 MB. The switches are configured with the CE marking threshold set to 90 KB. In this experiment, servers on one ToR initiate 1 MB flows randomly to any other server in the network with exponential inter-arrival times at a rate that cumulatively amounts to 20%, 40%, or 60% average utilization across the bisectional links. We initiate a total of 1.2 million flows, and wait for all flows to finish. We use the default TCP re-ordering threshold of 3, and monitor the out-of-order delivery numbers to ensure that FlowBender does not introduce undesired CPU (processing) overhead. To reconfirm that FlowBender does not lead to any abnormal packet re-ordering activity, we re-ran our experiments with a TCP re-ordering threshold of 30, and we did not see any noticeable difference in performance.

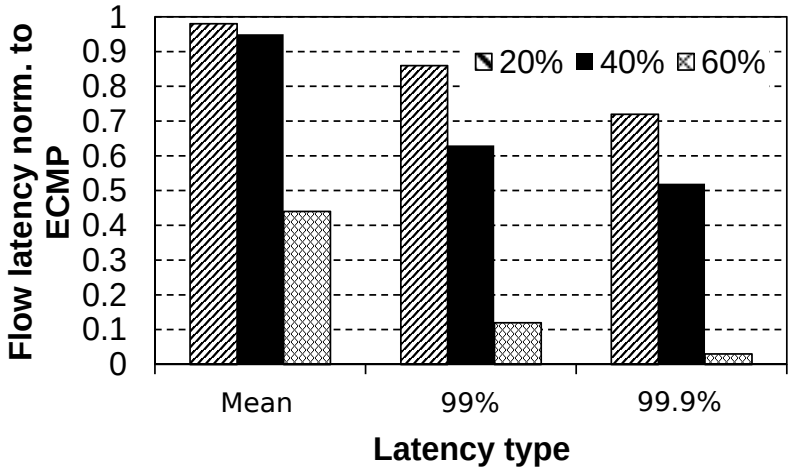


Figure 4.4: FlowBender's Latency Reduction at 20, 40, and 60% load (Bisectional).

In Figure 4.4, we show the mean, the $99^{th} percentile$, and the 99.9^{th} $percentile$ latencies along the X-axis, and FlowBender's completion time normalized to that of ECMP along the Y-axis. We use default parameter settings for FlowBender i.e., N = 1 and T = 5%. FlowBender improves the 99^{th} and $99.9^{th} percentiles$ by 15 - 26% and 34 - 45% respectively, in comparison to ECMP. At 60% load, FlowBender's flows finish more than twice as fast as ECMP on average, and 87 - 96% faster at the tail end. The qualitative results support the claim that FlowBender offers drastic improvement over static schemes like ECMP.

Decongesting HotSpots We now evaluate FlowBender's ability to decongest flows by re-routing them around hotspots. We initiate an all-to-all ran-

dom shuffle of 1MB TCP flows from one ToR to another (all in the same direction). The aggregate TCP traffic generates 14 Gbps from the sending ToR (arbitrarily spread across the 4 10 Gbps links). We also initiate 1 UDP flow between the same pair of ToRs, in the same direction as the TCP traffic, and rate limit it to 6Gbps. The purpose of the UDP flow is to create a static (asymmetric) hot spot along one of the four paths given that this flow will not be re-routed or load balanced by FlowBender. We denote the path which this UDP flow hashes on by U. Note that the aggregate TCP and UDP traffic on the four routes between the sending and the receiving ToRs amounts to 20 Gbps. Hence, in an ideal setting, one would wish that the 14 Gbps would have been equally split across the three paths other than U given that 14/3 Gbps is still less than UDP's 6 Gbps that was already routed on U. With ECMP, on one hand, U was unsurprisingly getting around quarter of the TCP traffic (14/4=3.5 Gbps) obliviously mapped to it, thus ending up with around 9.5 Gbps on average in total, driving that link practically unstable. FlowBender, on the other hand, succeeded in load balancing the traffic to a great extent with only around 1.5 Gbps of the 14 Gbps going on U. This experiment confirms FlowBender's ability to adaptively re-route around congestive hotspots in the network, and to respond to congestion created by non-TCP traffic. The above experiment is also interesting from a Weighted Cost Multipathing (WCMP) perspective (as opposed to ECMP) in the context of asymmetric topologies, where in order to reach a certain destination group, the viable ports at a switch are configured with different forwarding weights so as not to prematurely oversubscribe those paths with lower capacities. One of the challenges with WCMP is to be able to reflect the different weights of the forwarding ports accurately, which is highly dependent on how many entries the forwarding table can accommodate as per current ECMP implementations (i.e. larger tables can represent the different weights with higher granularity). The significance of this experiment is in how even if the forwarding weights suffer some inaccuracy because of the forwarding table being constrained to have few entries only (as is the case with our testbed and most of the commodity switches), FlowBender is able to dynamically re-adjust the traffic on the different available paths such that those with lower capacities are not severely congested (i.e. more robustness to forwarding weight misconfigurations or chip limitations).

Topological Dependencies Our results above are based on a topology with 8 and 4 different paths between any pair of pods or ToRs respectively. A question that commonly arises here is: how helpful would FlowBender be when the path diversity between any pair of pods increases? In other words, what role does FlowBender play if the port density of each switch is, say, doubled, together with the number of servers per ToR, while keeping the over-

subscriptions ratios the same (i.e. the path diversity quadruples)? The extent to which FlowBender helps clearly depends on the number of the available paths P, but it also depends on the number of those larger flows L that we're trying to spread out on those paths. More precisely, and particularly true in the limits, the performance improvement depends on the ratio $R = L/P$ of the two numbers as we show next, which is typically expected to remain constant given that as the bisectional capacity (i.e. P) is scaled up, the load, and hence, L would be also scaled up proportionally to maintain the same utilization. Considering the micro benchmark basic validation results discussed earlier, where FlowBender is shown to do a good job in evenly spreading the flows across the different paths, FlowBender's performance improvement amounts to how bad ECMP's flow distribution performance was in the first place. Given the oblivious nature of ECMP, the distribution of the number of large flows per each route is, in steady state, a very straightforward binomial distribution with a mean R and a variance $R(11/P)$, which is therefore not that different for a reasonably large P. For example, varying P from 8 to 32 would increase the variance by less than 11% only and hence would have a negligible effect in practice. In fact, we reran our All-to-All experiments with a different fan-out degree, and the performance improvement due to FlowBender was almost the same.

4.1.3 Further optimizations

Stability

When it comes to taking a rerouting decision, FlowBender inherits from the limitations of its rerouting component: ECMP. As described in Section 1.3, ECMP is just a hashing technique to select an egress link. When FlowBender takes a rerouting decision, it changes the `V` value, changing the value of the hash, randomly choosing a new path. Neither FlowBender or ECMP monitors the load on all available paths before taking a rerouting decision. Therefore, the new path it reroutes to may also be congested. If that happens to be the case because say of an incast episode or because the network is highly congested in general, FlowBender will trigger yet another path change, and the rerouting process may repeat. Accordingly, FlowBender can be extended to limit the number of path changes that could occur when the network is severely congested. More specifically, FlowBender can be constrained to switch paths for a maximum of S consecutive times before it goes into a locked state. In the locked state, it will pick one path out of the last S paths that had the lowest value of F, the fraction of ECN-marked ACKs, and will lock in to that path for the next U RTTs. At the end of the locked phase, FlowBender resumes business as usual, trafficking F and switching paths if it exceeds T for N consecutive RTTs. Note that if we were to choose S and

U as 5 and 10, respectively, with an N of 2 (which gives almost the same performance as the default $N = 1$ configuration), then we would be limiting the rerouting events to a maximum of 5 times in every $5 \times 2 + 10 = 20$ RTTs, thus significantly limiting the number of out-of-order packets that could be triggered by FlowBender.

Selective Rehashing

In our current implementation, we change the value of `V` for the flexible hashing input field obliviously once congestion occurs. Because of ECMP's design, changing the value of `V`, however, does not always mean that the route will change as this really depends on the hashing function. That said, given sufficient knowledge about the hashing functions in the fabric, one can avoid this artifact by having each flow precompute a number of potential values for `V` that would result in hashing to different paths. The process for precomputing such values might be very challenging and time consuming to perform for general hashing functions, but if the network operators choose their hashing functions in a way such that flipping one of the hashing inputs bits would result in a different hashing output (e.g. XOR hash functions), then this process would become much easier to perform. Alternatively, flows can correlate the different RTT estimates or values for F corresponding to different `V`'s in order to infer, with a high probability, how these `V`'s map to different paths and avoid choosing a redundant value for rerouting.

Proactive Probing and Route Caching

We have demonstrated the reactive version of FlowBender, where a flow is rerouted only once congestion has been detected. Of course, one might argue that FlowBender is already quite prompt in rerouting when congestion arises, given its low congestion detection threshold `T`, but the load balancing performance could be still further improved by allowing a flow to proactively probe across the different paths. By probing we mean that a flow can periodically send a few of its packets with different `V` values, keeping track of their sequence numbers ranges, and check once they have been acked which of them have experienced congestion. One of those `V`'s corresponding to probe packets which did not seem to be congested at all could be proactively selected as the basic `V` once F has exceeded a threshold T0 smaller than T. Alternatively, a flow may attempt to keep track of some of those better `V`'s to hash upon once congestion occurs, instead of choosing `V` obliviously, or might simply blacklist those highly congested `V`'s and avoid revisiting them until sufficient time has elapsed.

4.1.4 Conclusion

In this section, we described a new load balancing mechanism called FlowBender. The main motivation for introducing FlowBender is to overcome the limitations of oblivious hashing schemes such as ECMP, prominent in today's datacenters, without suffering from high packet re-ordering or requiring custom hardware changes and complicated host mechanisms that could offset any potential benefits. To summarize, the key strengths of FlowBender's design are that it:

- Requires no changes to switch hardware (silicon).
- Amounts to only 50 lines of kernel code change.
- Requires simple re-configuration to ECMP hash functions (a handful of commands).
- Substantially outperforms ECMP and matches the performance of other more complex schemes.
- Incorporates robust end-to-end congestion notifications (ECN) and failure signals (Timeouts).
- Reroutes at the Round-trip Time (RTT) granularity and recovers from link failures essentially within an RTO.

Our evaluation shows that it cuts the flow completion tail latencies by around 40% relative to ECMP's for large flows.

4.2 Leveraging IPv6 Segment Routing and eBPF to efficiently steer transport flows

The advent of IPv6 Segment Routing and eBPF, opens new possibilities for network engineers. While IPv6 Segment Routing enables network operators to explicitly route packets in their networks, eBPF allows to efficiently manipulate the network stack with a very small overhead. As shown in Section 4.1 it is complicated for transport protocols to leverage the path diversity when congestion or other problems occurs. While FlowBender allows to switch paths when congestion occurs, it is more complicated to switch to a path that is known to be less congested. This comes from the decoupling between the transport layer and the network layer. In the OSI model, the transport protocol has no precise knowledge of the path it is currently using. The same applies to the network layer that has few to no knowledge about the internal state of the transport protocol. While this design choice is interesting because it allows to easily combine different transport and network protocols, it clearly limits the transport protocol's possibilities when it experiences problems. With this in mind, a naive solution would be to replicate the transport protocol's internal state (RTT, number of loss, ...) inside the network layer. However, this naive solution is a cross-layer violation that would make compatibility between protocols difficult. In this section, we propose to combine eBPF and IPv6 Segment Routing to efficiently tackle this problem. With eBPF, we can efficiently monitor the transport protocol's internal state (e.g its number of re-transmissions). With this information combined with IPv6 Segment Routing's capability of routing traffic, we can change the path used by a flow when the transport protocol is experiencing problems on that path. This allows to use the transport protocol information to make routing decisions without committing an undesirable cross-layer violation.

This Section is organized as follows. First, we desribe a few use cases that could benefit from this combination in Section 4.2.1. In Sections 4.2.3 and 4.2.2 we describe our architecture and its building blocks. Then, in Section 4.2.4 we demonstrate the feasability of our approach in simple emulated scenarios. In Section 4.2.5 we describe the next steps of our solution.

4.2.1 Use cases

Re-routing flows using IPv6 Segment Routing

In Section 4.1, we presented FlowBender, an efficient solution to reroute flows when they use a congested path. One of the limitations of FlowBender is that it inherits from the limitations of its rerouting component: ECMP. When it detects that a flow is using a congested path with ECN, FlowBender changes

the flexible field V resulting in a new ECMP hash. However, there is no guarantee that this new path will be different, disjoint and not congested. In that case FlowBender tries again with another new path. To mitigate this problem, we propose to use IPv6 Segment Routing. FlowBender's limitation comes from the fact that it has no knowledge of the path used by the traffic. With IPv6 Segment Routing however, the path is explicitly specified in the SRH. By leveraging this capability, we can react to congestion by switching to a path that is different, disjoint and not congested. If no such path is available, we avoid an unnecessary switch of path.

Multi-homed hosts

While mobile phones have been offering 4G and WiFi for years, multi-homed hosts are getting more and more deployed in different environments. In order to offer higher bandwidths to their clients, ISPs have started to deploy hybrid access networks [Fab16], i.e. networks that combine different access links such a xDSL and LTE. In one deployment, described in [LHZ+17], a hybrid CPE router with xDSL and LTE is connected to an aggregation box with GRE Tunnels. The tunnels ensure that the packets sent by the hybrid CPE are routed to the aggregation box that reorders them. As shown in previous sections, multi-homed hosts are also largely deployed in datacenters. While Multipath TCP can be a good solution to leverage multiple interfaces, it requires important modifications to the networking stack. In this context, IPv6 Segment Routing can be used to replicate some of Multipath TCP's features at a lesser cost in terms of modifications. While bandwidth aggregation would be difficult to achieve because of the reordering cost at the receiving host, IPv6 Segment Routing can be used to steer specific flows through a specific interface, reduce the delay by using the lowest-RTT path or for failover purposes. Our solutions leverage the multiple interfaces without needing to significantly change the host's networking stack.

Limiting the impact of Heavy Hitters (elephant flows)

In Datacenters, a few long flows contribute to an important part of all the traffic [BAM10]. These long flows are often called heavy-hitters or elephant flows. While these "elephant" flows only account for a fraction of the flows, they are an important part of all the traffic in terms of bytes. On the other hand, a "mouse" flow is a short flow that typically serves a Remote Procedure Call (RPC). The mice flows only account for a small fraction of the traffic (in terms of bytes), but represent the largest number in terms of flows. A mouse flow is typically a short request/response query and is thus sensitive to latency. Because responses to complex queries are constructed by aggregating the results from several mice flows, the tail latency of the individual flows

directly affects responses time and quality. In today's networks, the problem appears when one tries to combine mice and elephant flows on the same path. The elephant flows are going to consume most of the available bandwidth, negatively impacting the latency of the mice flows. While solutions like the `hhf-qdisc` [qdisc-hhf] that uses a multi-stage filter [EV02] to detect the elephants are already available in the Linux Kernel, they solve the problem by giving priority to the latency sensitive traffic over the bulk one. With our architecture, we take a different point of view. Instead of trying to limit the traffic of the elephants, we propose to avoid the problem by detecting the elephants with eBPF and to route them on a separate path using IPv6 Segment Routing. This allows network operators to design networks with, mice dedicated low-latency links that do not need an important quantity of bandwidth and elephant dedicated links with high bandwidth and less requirements in terms of latency.

4.2.2 Building blocks

Linux control groups (cgroups)

The Linux control groups (cgroups) provide an interface to manage the behavior of processes. Cgroups can be used to limit or prioritize the access to resources (cpu, memory,...) and to account for the group's usage of said resources. All of the processes of a cgoup share the same limitations.

extended Berkeley Packet Filter (eBPF)

eBPF (for *extended Berkeley Packet Filter*), is a general-purpose virtual machine that is included in the Linux kernel since the 3.15 release. This virtual machine supports a 64 bits RISC-like CPU [BDK18] which is an extension of the BPF virtual machine [MJ93]. It provides a programmable interface to adapt kernel components at run-time to user-specific behaviours. While solutions such as [SCP+16, MZK+17] use P4 [BDG+14] to achieve data plane programmability, they are limited by the fact that P4 relies on specific hardware (and/or compiler), while eBPF targets a general purpose CPU. The LLVM project [LLV18] includes a BPF backend, capable of compiling C programs to BPF bytecode. eBPF bytecode is either executed in the kernel by an interpreter or translated to native machine code using a Just-in-Time (JIT) compiler. Since the eBPF architecture is very close to the modern 64-bit ISAs, the JIT compilers usually produce efficient native code [BL18].

eBPF programs can be attached to predetermined hooks in the kernel. Several hooks are available in different components of the network stack, such as the traffic classifier (`tc`) [Bor16], or the eXpress Data Path (XDP) [BX18], a low-level hook executed before the network layer, used e.g. for DDoS mit-

igation. When loading an eBPF program into the kernel, a verifier first ensures that it cannot threaten the stability and security of the kernel (no invalid memory accesses, no infinite loops, ...). The eBPF program is then executed for each packet going through the datapath associated to its hook. The program can read and, for some hooks, modify the packet.

eBPF programs can call *helper functions* [BH18], which are functions implemented in the kernel. They act as proxies between the kernel and the eBPF program. Using such helpers, eBPF programs can retrieve and push data from or to the kernel, and rely on mechanisms implemented in the kernel. A given hook is usually associated with a set of helpers.

There are two practical issues when developing eBPF programs. The first is how to store persistent state and the second is how it can communicate with user space applications. State can be kept persistent between multiple eBPF program invocations and shared with user space applications using *maps*. Maps are data structures implemented in the kernel as key / value stores [BX18]. Helpers are provided to allow eBPF programs to retrieve and store data into maps. Several structures are provided, such as arrays, hashmaps, longest prefix match tries, ...When processing packets, if information needs to be pushed asynchronously to user space, *perf events* can be used. Perf events originate from Linux's performance profiler `perf`. In a networking context, they can be used to pass custom structures from the eBPF program to the perf event ring buffer along with the packet being processed [BP18]. The events collected in the ring buffer can then be retrieved in user space. These mechanisms allow stateful processing and a user-space communication that would be difficult to achieve with P4. Finally, a lightweight tunnel infrastructure named *BPF LWT*, provides generic hooks in several network layers, including IPv6 [BL16]. This LWT enables the execution of eBPF programs at the ingress and the egress of the routing process of network layers, but is unable to leverage the specificities of IPv6 Segment Routing.

eBPF programs types The Linux kernel supports several eBPF program types. The program type determines several properties of the program:

- **Where** the program can be attached (ingress, egress, ...)
- **Which kernel functions** the program might call
- Whether packet data can be directly **accessed and modified**

In our architecture, we use two types of eBPF programs :

- `BPF_PROG_TYPE_CGROUP_SKB`: this program type acts as a packet filter. Depending on the path where it is attached, this program will be called for every packet received or emitted. When called, this program

type receives a reference to a socket buffer (`skb`). It can inspect every received or emitted packet but cannot modify socket options.

- `BPF_PROG_TYPE_CGROUP_SOCK`: this type of program is called by the kernel during certain events. It is allowed to modify socket options. Contrary to its `CGROUP_SKB` counterpart, this type of program is not called for every packet, but only when one of the supported events happens. In the Linux kernel, such events are when a connection is established, when the TCP state changes, or when a skb is re-transmitted,... This type of program does not receive a reference to an skb, but only to the socket concerned by the event.

IPv6 Segment Routing

Described in Section 1.5, IPv6 Segment Routing is used to steer packets through an ordered list of segments. The IPv6 flavor of Segment Routing (SRv6) leverages a dedicated IPv6 routing extension header, named Segment Routing Header (SRH). Each segment is an IPv6 address representing a node or link to traverse. The SRH thus represents a path to follow in the network. In the Linux Kernel, it is possible to steer a TCP connection through a certain path by attaching an SRH to the TCP socket. This can be done by using the `setsockopt` function.

Software Resolved Networks (SRN)

Many entreprises are seduced by Software Defined Networks (SDN) [M+08, CFP+07, KRV+15a] which promise to simplify the management of their networks. Software Resolved Networks (SRN) [LJC+18] instantiate this SDN vision by using IPv6 Segment Routing in enterprise networks. Like SDNs, SRNs use a controller that manages the network resources. As in SDNs, the presence of the controller simplifies the management of the network and allows the operator to better control the available resources.

However, there are several differences between SRNs and SDNs. First, SRNs leverage IPv6 Segment Routing in the dataplane and the SRH to control the flow of packets through the network. This reduces the amount of state required on the routers in contrast with Openflow-based SDNs. Second, applications can interact explicitely with the controller to indicate the requirements for their flows. The controller responds to these requirements by returning a SRH for a path that meets them.

In SRNs, the controller is co-located with the entreprise DNS resolver and hosts use the DNS protocol to interact with the controller/resolver. When an application initiates a conversation, it performs the following operations. First, it issues a DNS request to resolve the DNS name of the server and adds its requirements. Then, the controller chooses a network path that meets

those requirements. The controller can use any optimization algorithm to select this path. Once the path is chosen, it is transformed into a list of SRv6 segments. Different path selection algorithms can be included in the SRN controller. The controller then sends back to the endhost a DNS response containing the server IPv6 address and the SRH corresponding to the selected path. Finally, the endhost attaches the SRH to each packet of the connection.

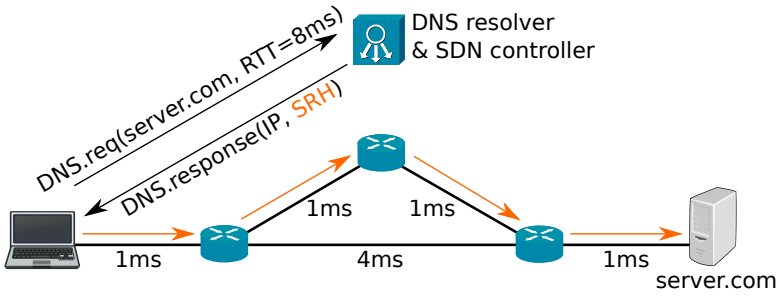


Figure 4.5: Illustration of path selection in SRN.

Figure 4.5 shows a Software Resolved Network. All its links have an IGP weight of 1. In a traditional IPv6 network, the application flows have to follow the shortest network path. In this example, the application wants a RTT of maximum 8 ms and the shortest path has a RTT of 12 ms. The controller selects the upper path and returns its SRH to the endhost. Additional details about SRNs and the realisation of its controller may be found in [LJC+18].

4.2.3 Architecture

Our architecture, illustrated in Figure 4.6 is built around the building blocks described in the previous section.

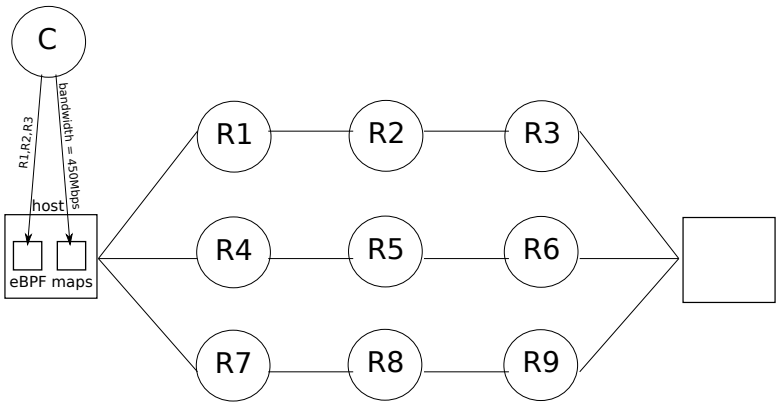


Figure 4.6: High level architecture.

The starting point is that we regroup the processes of the applications into `cgroups`. By doing so, it is possible to create a cgroup having the expected network behavior. For instance, bulk-transfer applications might be put in a group and latency-sensitive applications in another. This allows us to apply different policies depending on the application. The policy applied will depend on which eBPF programs is attached to the cgroup. To manage the communications between the controller and the host, we add a host daemon. This host daemon establishes an out-of-band connection with the controller. This connection is allows the controller to send updates about the path. Once the host daemon receives an update from the controller, it updates the relevant fields in the host's eBPF maps. In a simple scenario where an application wants to connect to a server, the application first queries the controller [LJC+18] about the different available paths to reach this server. With the help of the host daemon, the controller updates the local eBPF-Maps of the host with the available paths for this destination and their status. At this point, the host is aware of the available paths, and the rest of the process is at the host level. Figure 4.7 shows the process at the host level. To

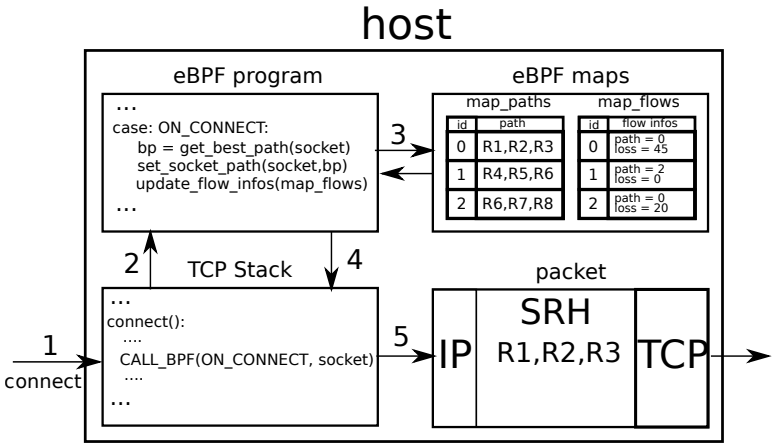


Figure 4.7: Host processing.

explain Figure 4.7, we consider a simple scenario in which the host wants to establish new connections on the best available path. In this scenario: (1) the application uses the `connect` system call. (2) the TCP stack executes the `connect` function, and calls the eBPF program attached to its cgroup (if any). (3) The eBPF program selects one of the paths previously filled by the controller in its eBPF map. (4) the eBPF program calls our implementation of `bpf_setsockopt` to attach this path to every packet sent on the socket. (5) the TCP stacks sends the packets through the network. At this point the eBPF program is no longer necessary in this simple scenario. The SRH being associated with the socket, the stack will insert it with every packet.

While in the previous example, the eBPF program is triggered by the `connect` system call, there are several ways to trigger an eBPF program in our architecture.

Reacting to kernel events

A first way of triggering an eBPF program is an event in the networking stack. An event is triggered when some conditions are fulfilled and detected by the networking stack. There are already several events defined in the Linux kernel. These are called when:

- a `connect` system call is issued;
- a connection is established;
- a retransmission timer (RTO) has expired;
- a skb is retransmitted;
- TCP changes state.

When a kernel event happens, the TCP stack might call a `BPF_PROG_TYPE_CGROUP_SOCK` program. For instance, when a re-transmission occurs the kernel will call the associated program. It can then decide if a change of path is needed or not.

An event should typically be added for very simple case and thus does not require more than a few lines of code in the kernel. In our implementation, apart from the `bpf_setsockopt` allowing `BPF_PROG_TYPE_CGROUP_SOCK` programs to set an `SRH`, we added events for cases such as when an ECN marked packet is received or when a packet is transmitted.

Reacting to an incoming/outgoing skb

When a packet is transmitted or received, a `BPF_PROG_TYPE_CGROUP_SKB` program might be called before the packet reaches the TCP/IP stack. This program has access to the whole packet, but cannot modify the associated socket. Because this kind of program is meant to act as a packet filter, it has to return a "verdict" indicating whether this packet must be dropped or not. Because it has access to the "raw" packet, this kind of program might me used for accounting or filtering purposes.

Reacting to other eBPF programs

Another way of triggering an eBPF program might be another eBPF program. In some cases, one could want to modify some socket option when a certain

kind of packet is received. This cannot be done by a BPF_PROG_TYPE_CGROUP_SKB program alone because it does not have write access to the socket. This cannot either be done by a BPF_PROG_TYPE_CGROUP_SOCK program alone because it does not have access to specific packets. A simple way of solving this problem could be to add an event in the kernel in order to call a BPF_PROG_TYPE_CGROUP_SOCK when such packet is received. However, if the information needed to trigger the event lies in the payload of the said packet, the processing might be complex. Let us take the example of a specific ICMP message received from a router in the network, informing that the current path is congested and containing a new path to use. Modifying the kernel to implement such a complex analysis is neither easy nor desirable. To support that case, we use a BPF_PROG_TYPE_CGROUP_SKB to parse the packet and extract the information. When the specific ICMP message is detected by this program, it extracts its content and places it in an eBPF map (for instance, replacing the current SRH by the one in the ICMP packet). The program then forwards the packet to the stack. Later (this can be just after the reception of the packet if program our called for every packet is enabled), when an event is triggered for the socket concerned by this message, the BPF_PROG_TYPE_CGROUP_SOCK program will be called. This program can then inspect the map to check if any of the path has been modified by another program. To detect such changes, we use a "DIRTY BIT" that's set by the program modifying the map. If such change is detected, the BPF_PROG_TYPE_CGROUP_SOCK can then proceed to modify the path for the socket by calling our dedicated `bpf_setsockopt`.

The need of a controller

At this stage, it is important to specify that if the controller described in this section is an important part of our architecture, it is however not strictly necessary. The controller is important because it allows to have a global view of the network, computes disjoint paths [AVBD18] and to receive updates on some metrics. It is however still possible to use our IPv6 Segment Routing-eBPF architecture without a controller for certain metrics if the different paths between two hosts are statically defined. Some metrics are difficult to compute by the TCP/IP stack. The bandwidth available on a path for instance does not only depend on the path usage by the sole host our eBPF program is running on. Other hosts might congest the path, so a program that selects the path by checking the available bandwidth needs a controller. Other metrics however can easily be measured by a host without the help of a controller. The Round-Trip-Time (RTT) for instance is already measured by the TCP stack without any help from a controller. For instance, a program that selects a path based on the RTT does not need a controller to update this met-

ric. Every RTT, this program is called by the stack and updates the eBPF-Map containing the path with the RTT the TCP stack computed. If the RTT is over a certain threshold, the program can lookup an alternative path with a lower RTT in the map. To ensure that the RTT of the alternative path is accurate, it is not strictly necessary to rely on continuous probing. While continuous probing could be an interesting technique it complexifies the implementation and sends unnecessary traffic onto the network. To ensure that the RTT of the alternative path is accurate, we rely on other connections. Indeed, if the host is actively communicating some other connection will use these paths at some point, and thus update the RTT observed for this path. A limitation of this approach is that we have to make sure that no path stays unused for too long. To address this, if a path has not been used for a certain amount of time, a new connection will be steered on this path. This connection will then estimate the RTT, update it in the map, and move if it is over the threshold.

4.2.4 Evaluation

In this section, we evaluate our architecture by running emulated scenarios with mininet [mininet]. The purpose of this section is to evaluate the capability of using IPv6 Segment Routing and eBPF to implement our architecture. The performances of the controller are therefore out of the scope of this section and will be evaluated in a different publication.

Reacting to ECN

In Section 4.1, FlowBender proved to be an efficient solution to re-route flows based on the ECN feedback, its main limitation being its dependence on ECMP. For this evaluation, we use a `BPF_PROG_TYPE_CGROUP_SOCK` program that, like FlowBender, reacts to congestion detected by ECN. We consider the topology illustrated in Figure 4.8. When it detects congestion by receiving ECN marking, this program tries to move the flow to a path where no congestion event has recently been observed. When a congestion event occurs, the program remembers the timestamp and changes the path if possible. After changing the path, a flow has to wait for `N` seconds before changing it again.

In this topology, there are 3 paths available between the client and the server, and each link is configured at 100Mbit/s. The scenario consists in a client process running in a cgroup with our `BPF_PROG_TYPE_CGROUP_SOCK` program. The client tries to send data to the server a fixed rate of 10 Mbits/s. After a delay of 5 seconds, we create congestion between two of the routers (R1-R2) of the currently used path by running an uncapped iperf [iperf].

Figure 4.9 illustrates this scenario with our eBPF program deactivated. As expected, after 5 seconds, our 10Mbit/s flow throughput drastically drops to less than 4Mbit/s and then oscillates between 4Mbit/s and 7Mbit/s for the

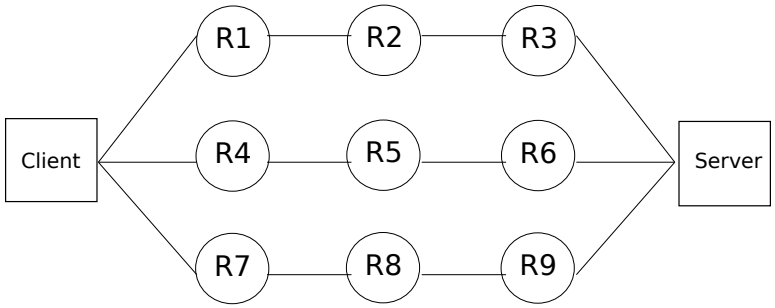


Figure 4.8: Topology of our mininet network.

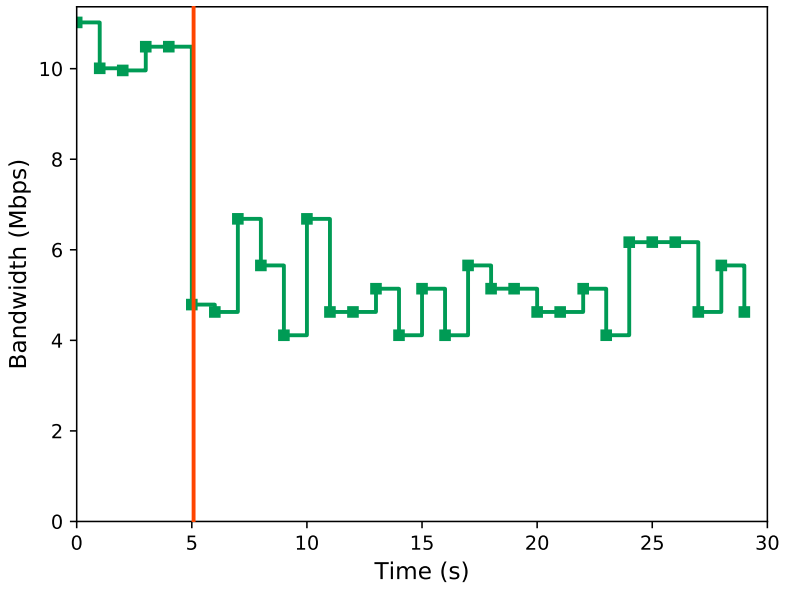


Figure 4.9: Observed goodput of a TCP flow without our eBPF solution.

remainder of the connection. This is expected and is due to the congestion created on the path.

Figure 4.10 compares the same scenario but with our eBPF program enabled. In this case, after 5 seconds there is no observed drop in the throughput of the client. This is due to the fact that upon reception of the ECN mark, our eBPF program moved the flow to one of the two other paths that was not experiencing congestion.

This simple evaluation shows that it is possible and practical to replicate FlowBender's behavior, with the advantage that the flow moves to a path that is **known to be different and disjoint**. In this topology, the basic FlowBender incurs a 33.3% risk of staying on the same path due to ECMP generating a new hash that ended up on the same path.

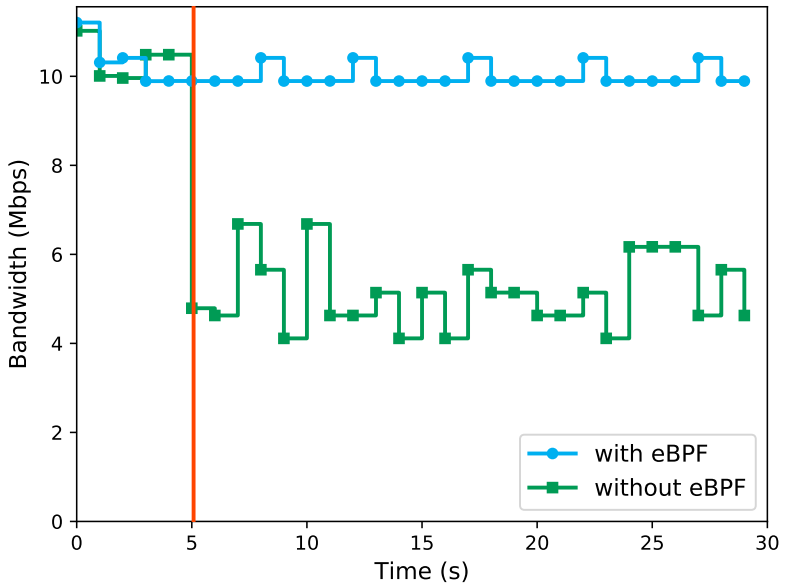


Figure 4.10: Comparison of the observed goodput of a TCP flow with and without our eBPF solution.

In a case where the second path (and/or the third path) is also experiencing congestion, our program can rely on the fact that other connections having experienced congestion on that path will update the map with the timestamp of the congestion event. In the case where that path has not been used by this host for a long time, there is a risk of moving a flow to that path only to find it experiencing congestion. To limit this possibility the solution is to rely on the controller updating the map when other hosts experience congestion on that path.

Limiting the impact of elephant flows

For this evaluation, we use mininet and the topology illustrated in Figure 4.11. In this topology, a client has two 100 Mbit/s paths to a server. The upper link represents the LTE interface and is configured with a delay of 20 ms each way. The other link represents a xDSL link and is configure with a delay of 40 ms each way.

In this scenario, we use the LTE link for latency sensitive traffic and the xDSL link for bulk traffic. To simulate latency sensitive traffic, we use `apache benchmark`[AB] running HTTP requests of 1KB. To simulate bulk traffic we launch 3 unlimited instances `iperf`. The algorithm of our `BPF_PROG_TYPE_CGROUP_SOCK` program is simple : each flow starts as a mouse/latency sensitive, thus using the LTE link. If a flows transfers more than 1 MB over a

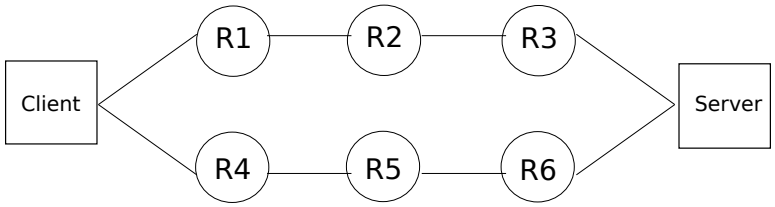


Figure 4.11: Topology of our mininet network with two links.

period of 1 second, it is identified as elephant/bulk flow and is moved to the xDSL link.

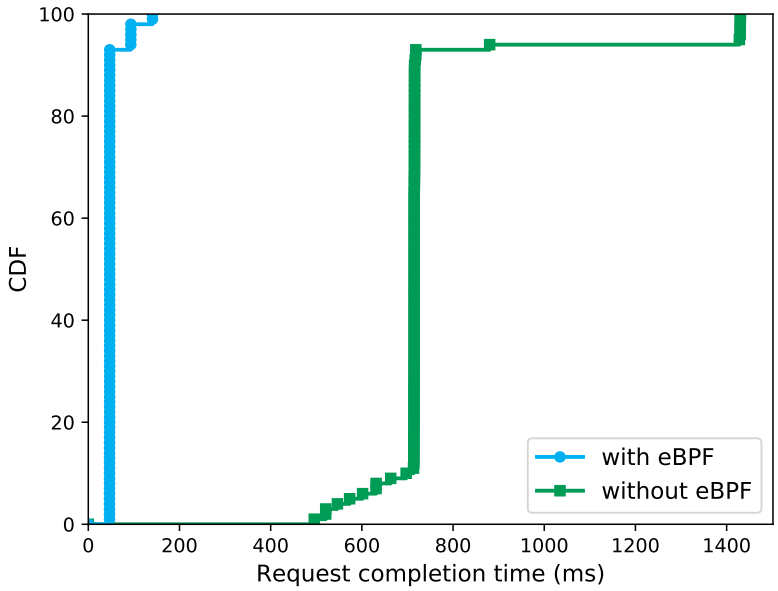


Figure 4.12: Latency of short flows with and without our eBPF program.

Figure 4.12 shows the latency of the latency sensitive traffic as reported by apache benchmark. Without our eBPF program, the mice experience a latency of 715 ms on average. This is caused by the congestion created by the elephant flows on the LTE link. With our eBPF program enabled however the mice experience a latency of 57ms, more than 15 times faster. This is due to the fact that the bulk traffic has been moved to the xDSL link. This simple experience shows that eBPF and IPv6 Segment Routing can be efficiently used to identify elephant flows and move them to a dedicated link in order to void any impact on latency sensitive traffic.

4.2.5 Future Work

Now that we established that eBPF and IPv6 Segment Routing can efficiently be used to steer flows over the network, our next step is to implement more eBPF programs that cover more use cases. While we presented only 3 use cases in this section, the flexibility of eBPF allows to envision many more use cases. The remainder of the future work will consist in evaluating the controller's performance and build more elaborated uses cases upon it. During our next steps, we will also evaluate our architecture using other transports protocols like QUIC or Multipath TCP. In the case of Multipath TCP, our architecture already works because each of Multipath TCP's subflows is a TCP socket. However, our current implementation does not take into account that these TCP sockets actually belong to the same Multipath TCP connection. With some small modifications, our architecture could take this information into account and offer Multipath TCP path manager new possibilities, like opening two new subflows on two totally disjoint paths. This is impossible at this time and often results in two subflows sharing the same bottleneck. With our architecture, the Multipath TCP Scheduler and Path Manager could make more informed decisions.

4.2.6 Conclusion

The objective of this section was to design a new architecture to "fill the gap" between the transport and the network layer. By leveraging IPv6 Segment Routing routing potential and eBPF's ability to access TCP's internal state, we demonstrated that it is possible to improve transport protocol performances by selecting a path according to the transport protocol's status. We modified the Linux kernel to add new events and the possibility for eBPF to attach an SRH to an existing TCP connection. We implemented eBPF programs to emulate to behavior of FlowBender and solve the problem of the elephant flows in a new way.

Chapter 5

Making Multipath TCP friendlier to Load Balancers and Anycast

5.1 Introduction

During the last years, several use cases have emerged for Multipath TCP [BS16]. Apple uses Multipath TCP on all its tablets, smartphones and laptops to support the Siri voice recognition application. Apple also uses Multipath TCP for its audio streaming service Apple Music, and has opened its Multipath TCP API to other applications. In this use case, Multipath TCP provides very fast failovers when a smartphones leaves or enters the coverage of a WiFi access point. A second use case for Multipath TCP is bandwidth aggregation. In Korea, high-end smartphones include Multipath TCP to bond the bandwidth of their WiFi and cellular interfaces and achieve higher throughputs [Seo]. Finally, network operators have started to deploy Multipath TCP proxies to bond xDSL and LTE networks in rural areas [BS16, BBG+19].

Given that the Multipath TCP specification was published in 2013, the Apple deployment that began in September 2013 is the fastest deployment of a TCP extension [Fuk11]. Despite this fast start, Multipath TCP is still not widely supported by servers [MHFB15]. There are several reasons for this limited deployment. On the client side, the implementation in the Linux kernel [PB+] is not included in the official kernel. On the server side, the deployment of Multipath TCP is hindered by technical problems [PGF15]. To leverage the different paths available in the network, operators frequently use load-balancing techniques. This technique is also heavily used for servers [Datanyze]. There are basically two families of load balancers: the stateless and the stateful load balancers. Stateful load balancers maintain state

for each established TCP connection and load balance them among different servers. Some of these load balancers have been upgraded to support Multipath TCP [bigip, Netscaler]. However, the state maintained by these load balancers limits their scalability. This is the reason why large content providers prefer to deploy load balancers [EYC+16] that store as less state as possible. Those load balancers operate on a per-packet basis and take their load balancing decision based on fields of the IP and TCP headers (e.g. the 5-tuple that uniquely identifies each TCP connection). Multipath TCP breaks this assumption and forces to rethink the operation of stateless load balancers. Several researchers have proposed modifications to load balancers to support Multipath TCP [OR16, LD16]. We discuss them in details in Section 5.2.

In this Chapter, we view the load balancing problem from a different angle, closer to one of the key ideas of Multipath TCP's design: having a protocol that is deployable in today's networks, without changing them. Instead of changing the load balancers to support Multipath TCP, we propose to slightly change Multipath TCP to be compatible with existing load balancers. This enables Multipath TCP to be used in any environment containing load balancers and is a much simpler deployment path than changing load balancers. Our modification is simple since it relies on a single bit in the `MP_CAPABLE` option exchanged during the three-way handshake. It has been integrated in the forthcoming revision of Multipath TCP [FRH+19]. We implement it in the Linux kernel and demonstrate its performance based on lab measurements. Furthermore, we show that with this small modification it becomes possible to efficiently support anycast services over Multipath TCP. This opens new benefits for Multipath TCP in addition to the existing bandwidth aggregation and fast failover use cases.

5.2 Background and motivation

A key benefit of Multipath TCP is that a Multipath TCP connection can transport data over different paths. A typical example is a smartphone that wants to use both its WiFi and LTE interfaces to exchange data. As explained in Section 1.4.3, a Multipath TCP connection always starts with a three-way handshake like regular TCP connections, except that the `SYN` and `SYN+ACK` carry the `MP_CAPABLE` option. Considering our smartphone example, if the initial subflow was created over the cellular interface, then another subflow is created over the WiFi interface. Each subflow is created by using the TCP three-way handshake with the `MP_JOIN` option in the SYN packets. This option contains an identifier of the Multipath TCP connection to which the subflow is attached (called token in [FRHB13]) that is derived from the information exchanged in the `MP_CAPABLE` option and some authentication information.

5.2.1 Load balancing principles

A network load balancing infrastructure is typically composed of one or several load balancers located between the physical servers that host the content and the edge routers as shown in Figure 5.1. In this chapter, we focus on Layer-4 load balancers [LVS, EYC+16, Barracuda] that do not terminate the TCP connection unlike some Layer-7 load balancing solutions [NGI, Haproxy].

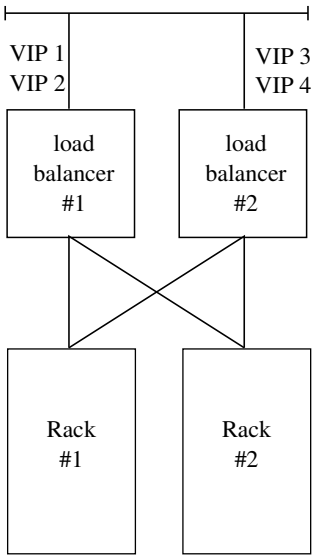


Figure 5.1: Typical deployment of Layer-4 load balancers.

Load balancers typically announce Virtual IP addresses (VIP). A VIP differs from a traditionnal IP address because it is not assigned to a single server. It usually belongs to a service whose content will be served by multiple servers located behind the load balancers.

When a client tries to connect to a service, it usually obtains the VIP via a DNS query and then sends packets to this address. When the first packet of a connection reaches the load balancer, the load balancer needs to select one of the service's servers for this particular connection. The specific algorithm used to select the best server is outside the scope of this chapter, but it is important to emphasize that once the server has been selected, all the packets belonging to this specific connection will be forwarded to that particular server. With regular TCP, load balancers usually extract the 5-tuple of the connection (protocol, source address, destination address, source port, destination port) from each received packet and assign each tuple to a specific server. Some load balancers simply forward these packets to the corresponding server [LVS] while others encapsulate the packet using Generic Routing

Encapsulation (GRE) [EYC+16] or other tunneling techniques. The main benefit of encapsulation is that the physical server does not need to be physically close to the load balancer.

This solution works perfectly with TCP and UDP because each TCP or UDP packet contains the five-tuple that identifies the flow. Multipath TCP unfortunately breaks this assumption. A load balancer should send all the packets that belong to a given Multipath TCP connection to the same physical server. Since a Multipath TCP connection is composed of different TCP connections, a packet can reach the load balancer via any of these TCP connections and thus via possibly different five-tuples. A Layer-4 load balancer cannot rely only on the information contained in this packet to determine the physical server that was selected for this specific Multipath TCP connection.

Several solutions have been proposed to reconcile Multipath TCP with load balancers. A first approach is to use stateful load balancers. Some commercial products [bigip, Netscaler] already support Multipath TCP and researchers have proposed stateful Multipath TCP capable load balancers [LD16]. Olteanu and Raiciu propose in [OR16] a modification to Multipath TCP that enables stateless load balancers to support Multipath TCP. Their solution relies on changing the TCP timestamp option. Instead of using this option to encode regular timestamps, they encode an identifier of the physical server behind the load balancer inside the low-order bits of the timestamp option [BBJS14]. Since clients always echo the timestamp option sent by servers, this enables the load balancer to receive in each packet an identifier of the physical server that needs to receive the packet. This solution has been implemented and tested in lab environment [OR16]. However, it suffers from three important limitations. First, load balancers and servers need to be modified to extract the information from the timestamp option. Second, this option, like any TCP option, can appear anywhere in the extended TCP header. This implies that a hardware implementation will be more complex than existing hardware solutions that simply extract the source and destination addresses and ports that are placed at fixed locations in all packets. Third and more importantly, there are various types of middleboxes that are deployed on the global Internet that change the values of the timestamp transported in TCP Options [HNR+11]. This solution is thus fragile in the presence of such middleboxes.

Stateless load balancers are much more scalable than stateful load balancers and large content providers want to continue to use stateless approaches for load balancing. In this chapter, instead of modifying the load balancer, a device that is usually hard to modify, we modify the protocol by slightly changing how addresses are advertised and used. Our choice is driven by the fact that Multipath TCP has been developed with a strong idea in mind: being deployable in today's networks (i.e without modifying the middleboxes). This

idea has deeply influenced the design of the protocol and is responsible for a big part of its complexity. Given that the IETF is currently finalising the revision of Multipath TCP [FRHB13] to publish it as a standard track document [FRH+19], this is the right time to propose such a modification.

5.3 Modifications to Multipath TCP

In a nutshell, our modification to Multipath TCP to support load balancers is to assign two addresses to each physical server: a VIP that is the load balanced address and a unique address that is assigned to each physical server. When a client creates a Multipath TCP connection to a load balancer, it uses the VIP and the load balancer forwards the packets to the selected physical server. The physical server advertises its unique address and the client immediately creates a second subflow towards this address. In this section, we provide an in-depth explanation of how we modified the protocol to support the new feature.

5.3.1 Restricting the initial subflow

Our first modification concerns the initial subflow that is created by the client. This subflow is created by sending a `SYN` packet that contains the `MP_CAPABLE` option. The format of this option is detailed in Figure 1.5. This subflow is established between the client address and the VIP served by the load balancer. As described previously, the main problem for Multipath TCP is that this VIP cannot be used to establish additional subflows. To support our architecture and avoid clients trying to establish subflows that could not be established, we modify the protocol to allow a host to be able to inform the other end that this specific address cannot be used to establish additional subflows.

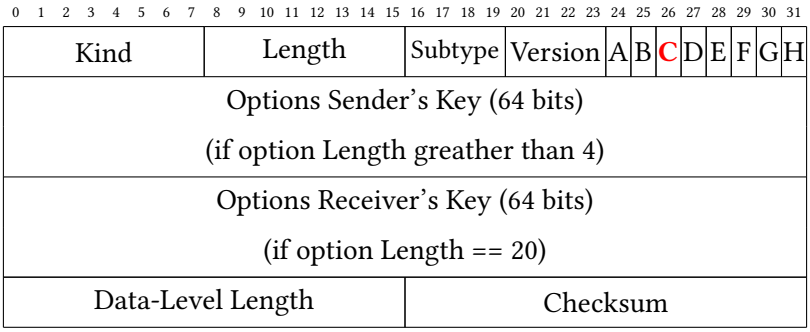


Figure 5.2: Revised Multipath Capable (MP_CAPABLE) Option (RFC6824bis).

This is done by adding a new "C" flag to the `MP_CAPABLE` option returned by the physical server. This change is shown in Figure 5.2 that can be compared with Figure 1.5. In the original format, the "C" flag was reserved for crypto algorithm negotiation, but was not used. We chose to use this flag to carry the *"don't join"* meaning. When the "C" flag is set, this indicates that the source address of the packet carrying this option (in this case the VIP address) cannot be used to create additional subflows. If a smartphone receives a `SYN+ACK` packet with the "C" flag set in response to a `SYN` packet sent over its cellular interface, it infers that it cannot create any additional subflow towards this address. To allow the protocol to know in which case this "C" flag must be set, we add a simple flag in the software that is used to configure addresses on the host. When the VIP is added to the host, it is possible to specify a *"no join"* flag for this specific address. This sole modification prevents the smartphone from creating a subflow that would not be correctly load balanced by the load balancer.

5.3.2 Using unique addresses

At this point, all the packets of the initial subflow will reach the load balancer and be forwarded to the physical server chosen by the load balancer for this connection. To benefit from Multipath TCP's features, the client must be able to establish additional subflows. If the client creates another subflow, the packets belonging to this connection must also reach the same physical server. For this, we configure each physical server with two addresses: the VIP and a unique (physical) address. Then, we leverage the existing `ADD_ADDR` option [FRHB13] and illustrated in Figure 1.10 to advertise it to the client. If the physical server advertises its physical address then the client will be able to create additional subflows towards this address which can bypass the load balancer.

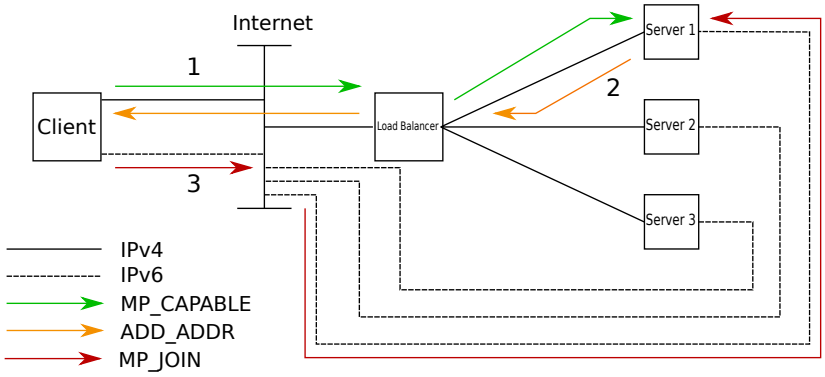


Figure 5.3: Description of our architecture.

This is illustrated in Figure 5.3 where: *(1)* the client connects to the VIP passing through the load balancer and the servers informs the client that this address cannot be used to establish new subflows; *(2)* the server advertises its physical IP address to the client; *(3)* the client connects directly to the server's physical address.

5.3.3 Reliable ADD_ADDR

If the "C" flag has been set in the MP_CAPABLE option, the client is prohibited from establishing any additional subflow until it has received the ADD_ADDR option that advertises the unique address of the server. Unfortunately, as explained in 1.4.5, according to the current Multipath TCP specification, the ADD_ADDR option is sent unreliably. This implies that any loss of the packet carrying this option could be problematic. In the best case, the client would be limited to the single initial subflow. In other cases, this could break the connection e.g. for a smartphone that moves away from the wireless access point used for the initial subflow.

To ensure that the ADD_ADDR is reliably transmitted we need to design an acknowledgment mechanism. The first solution, is implicit. When a server receives an MP_JOIN for an address that it just advertised this address can be considered as acknowledged. While this solution is sufficient in our case because the client will always establish a connection as soon as a new address is advertised, the specifications defines that the establishment of a new sublfow is at the client's discretion. To allow a generic extension to the protocol and ensure that addresses are reliably advertised, we proposed an echo mechanism for the ADD_ADDR option. The modified option is shown by Figure 5.4 which can be compared to Figure 1.10.

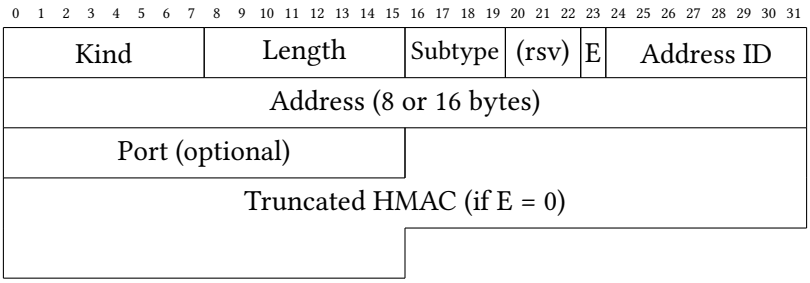


Figure 5.4: Modified Add Address (ADD_ADDR) Option (RFC6824bis).

In the newer version, we remove the IPVer field because the version of the IP protocol can be inferred from the length of the option. Instead, we add the "E" bit or "Echo" flag. This bit is reset when a host advertises an address. When a host receives an ADD_ADDR option with the "E" flag reset, it must

echo this ADD_ADDR option with the "E" flag set. This echoing serves as an acknowledgement of the ADD_ADDR option. This mechanism is illustrated in Figure 5.5.

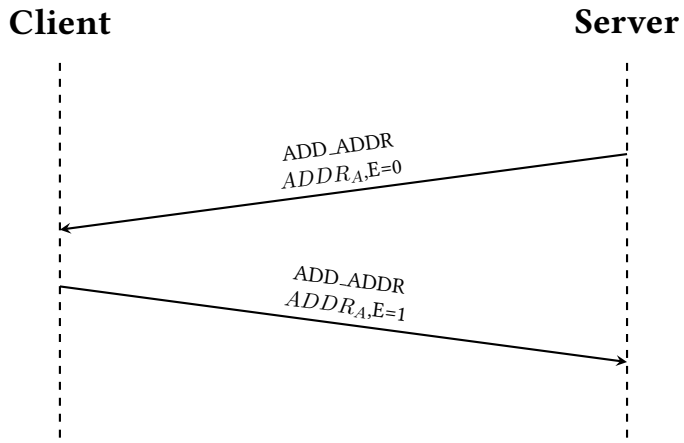


Figure 5.5: Address advertisement echo mechanism.

5.4 Use cases

In this section, we describe two use cases leveraging the protocol extension described in the previous section. The first one is that it becomes possible to place the load balancers complete off-path once the Multipath TCP connection has been established. The second one is that with our proposed extension it becomes possible to deploy anycast services over Multipath TCP, even in small networks.

5.4.1 Beyond Direct Server Return

Several deployment scenarios exist for load balancers. A simple approach is to place the load balancer in front of all the physical servers such that all the packets sent and received by the physical servers pass through the load balancer. This type of deployment is widely used when only a few physical servers are used. The main advantage of this deployment is that it is simple to deploy and operate. However, since all packets pass through the load balancer, it could become a bottleneck when the network load increases. This is illustrated in the left part of Figure 5.6.

Large web farms use a different approach to deploy their load balancers to support higher traffic loads. HTTP is highly asymmetrical. Most of the HTTP traffic is composed of the data packets that are sent by the physical

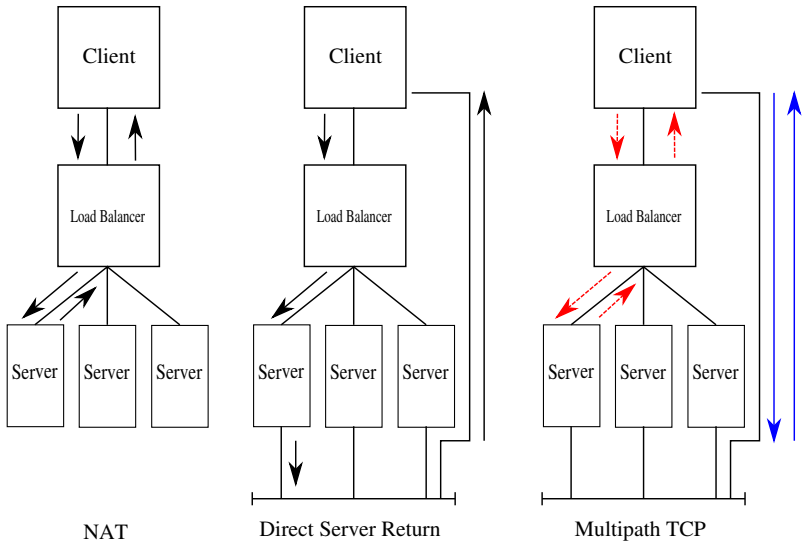


Figure 5.6: Different types of load balancer deployments.

servers towards the clients. The clients themselves only send the HTTP requests which are much less frequent. Many web farms leverage this traffic asymmetry by configuring the router/switch attached to the physical server to send the packets generated by those servers directly to the clients without passing through the load balancer. The packets sent by the clients (TCP acknowledgements and HTTP requests) still need to pass through the load balancer to be forwarded to the selected physical server. This deployment is illustrated in the center of Figure 5.6.

With our proposed modification to Multipath TCP, it is possible to go beyond Direct Server Return and completely bypass the load balancer for any type of TCP connection. The client establishes the initial subflow with the load balancer that forwards all packets belonging to this subflow to the selected physical server. The physical server advertises its address and the client creates an additional subflow towards this server address. All the packets sent to and from the physical server address automatically bypass the load balancer. Once the additional subflow has been established, the physical server can terminate the initial subflow so that no packet passes through the load balancer anymore. Storage services like Dropbox and Google Drive where the HTTP traffic is less asymmetrical could benefit from this modification. Several APIs have already been proposed and implemented to enable applications to control the Multipath TCP subflows [HB16, HDB+15]. This deployment is illustrated in the right part of Figure 5.6 where the red arrows (center) are related to the initial subflow, and the blue arrows (right) to the secondary subflow.

5.4.2 Supporting Anycast Services

There are three types of addresses that can be supported in an IP network: *(i)* unicast addresses, *(ii)* multicast addresses and *(iii)* anycast addresses. The unicast service is well-known. Multicast is outside the scope of this chapter. Anycast has been initially proposed by Partridge et al. in [PMM93]. Anycast applies to a network that contains several hosts that provide the same service. If each of these hosts is configured with the same anycast address, then when a client sends a packet towards the anycast address associated to the service, the network automatically forwards the packet to the closest host. Anycast has several appealing features such as its resilience to failure or its ability to minimize latency. Anycast is widely used to deploy DNS resolvers in ISP or enterprise networks [FHG13, AL06]. Given the privacy and security constraints of the DNS service, several researchers have proposed to run the DNS service above TLS and TCP instead of UDP [ZHH+15, HZH+16].

Unfortunately, it is difficult to use TCP servers with anycast addresses [NG14]. To understand this difficulty, let us consider the simple network topology shown in Figure 5.7. There are two anycast servers in this network shown as $Server$ in the figure. One is attached to router `R2` and the other is attached to router `R4`. Both advertise the same anycast address in the network. If the client attached to router `R1` creates a connection towards this anycast address, the resulting packets are forwarded to the server attached to `R2`. If the `R1-R2` link fails, the next packet sent by the client towards the anycast address will be delivered to the server attached to `R4`. Since this server does not have state for this TCP connection, it will send a RST packet to terminate it and the client will have to restart this connection.

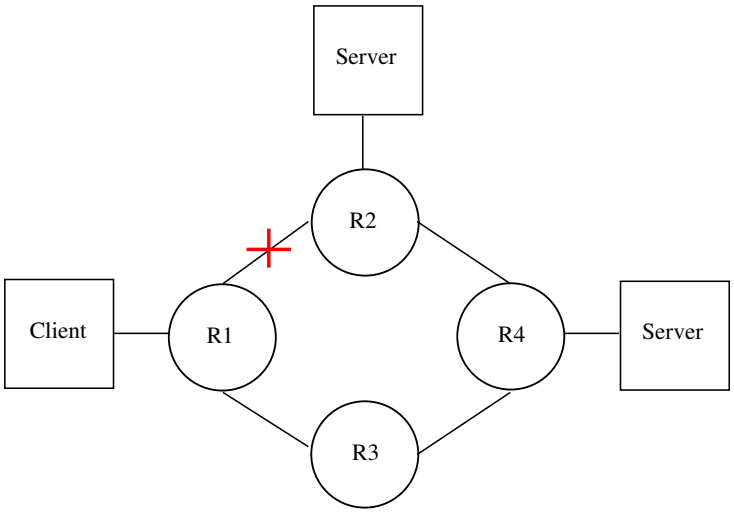


Figure 5.7: Anycast workflow.

Thanks to our proposed extension to Multipath TCP, it becomes possible to support anycast services. For this, each unicast server must be configured with two addresses: *(i)* the anycast address that identifies the service and *(ii)* the unique server address that identifies the physical server. Let us consider the same scenario as above. The client creates a Multipath TCP connection towards the anycast address. The network forwards the `SYN` packet to the server attached to router `R2`. This server accepts the Multipath TCP connection and replies with a `SYN+ACK`. The server then advertises its unique address over this initial subflow and then signals to the client to consider this subflow as a backup one. If link `R1-R2` fails, the packets of the initial subflow reach the server attached to `R4`. This server does not have state for this subflow and thus replies with a `RST` packet that terminates the initial subflow. This does not affect the other subflow that is bound to the unique address of the server attached to `R2`. The Multipath TCP connection with the server attached to `R2` continues without any impact on the client.

Anycast TCP services are typically deployed by associating a Fully Qualified Domain Name (FQDN) to each service and using the DNS server to spread the load among different servers. However, there are several situations where a DNS-based solution might not work. A first example are the DNS resolvers mentioned earlier [ZHH+15, HZH+16]. Those servers must be reachable via an IP address that is advertised by DHCP or through the IPv6 router advertisements. Another example are the different types of proxies that are being discussed within the IETF [BBG+19, Seo, BS16].

5.5 Performance Evaluation

To demonstrate the benefits of the solution described in the previous section, we first modify the reference implementation of Multipath TCP in the Linux kernel [PB+]. We then use this implementation to perform experiments in a lab with both load balancers and anycast services.

5.5.1 Implementation in the Linux kernel

The Multipath TCP implementation in the Linux kernel [Paa14] is divided in three parts. The first part includes all the functions that send and receive TCP packets. The second part is the path manager. This module contains the logic that manages the different subflow. Several path managers have been implemented [Paa14, BFM13]. The reference implementation contains the `full-mesh` and the `ndiffports` path managers. The `full-mesh` path manager is the default one. It tries to create a full-mesh of subflows among the addresses available on the client and the server. The `ndiffports` path manager was designed for single-homed clients and servers. On the client side,

```
WHEN A NEW CONNECTION IS ESTABLISHED:
/* Get the specific IP address */
ip_addr = GET_SERVER_IP()
/* Send an ADD_ADDR containing that
address to the client */
ADVERTISE_TO_CLIENT(ip_addr)
/* Change the first subflow to backup mode */
SET_BACKUP_MODE(get_first_subflow())
```

Figure 5.8: Pseudocode from our path manager.

it creates n subflows with different source ports towards the server. It was designed for the datacenter use case described in [RBP+11]. The third part is the packet scheduler that selects the subflow that will be used to transmit each packet.

We first add support for the "C" flag in the code that processes the MP_CAPABLE option described in Section 5.3.1. To support this flag, we had to modify the path manager used by the client to prohibit it form creating any subflow towards the destination address of the initial subflow.

Our second modification was to add the support of the "E" bit in the ADD_ADDR option as described in Section 5.3.3. We implemented it, by sending the ADD_ADDR option in every packet until the reception of the address acknowledgement (reception of an echo, with the"E" bit set to 1), making the transmission of the ADD_ADDR option reliable.

To support these two modifications, we have created a new path manager that is tuned for servers behind a load balancer. This path manager does not create any subflow, this is the standard behaviour of path managers running on servers. It advertises the unique server address on the initial subflow and then changes the priority of this subflow to become a backup subflow. Multipath TCP [FRHB13] defines backup subflows as follows : *path to use only in the event of failure of other working subflows.* This means that the initial subflow, that passes through the load balancer can still be used but the server encourages the client to use the subflows towards its unique address. An alternative would have been to reset the initial subflow, but this would have been less failure resilient. We have preferred to set the initial subflow in backup mode. The algorithm of this path manager is illustrated by the Figure 5.8.

These modifications represent approximatly 600 lines of kernel code, splitted into three patches (one by feature). 50% of the code lays in the path manager that can easily be plugged into the Linux kernel implementation.

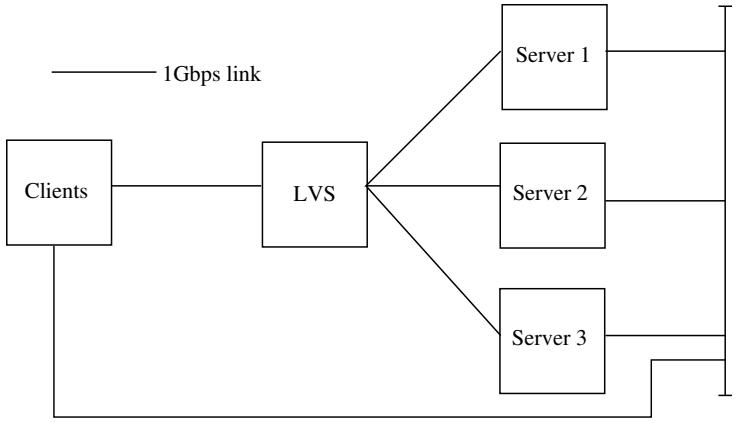


Figure 5.9: Evaluation setup.

5.5.2 Layer-4 load balancers

To evaluate the performances with load balancers, we use the network shown in Figure 5.9. The client is a 2 GHz AMD Opteron 6128 with 16 GB of RAM running Debian Linux with our modified version of the Multipath TCP kernel. This version is based on Linux kernel version 4.4. The server uses the same hardware configuration and runs lighttpd version 1.4.35 with the same kernel as the client.

The client accesses the web server via a VIP. This VIP is attached to the load balancer. Our load balancer runs on a 2.5 GHz Intel Xeon X3440 server running Linux Virtual Server (LVS) [LVS] configured in NAT mode. We use 1 Gbps Ethernet links between the load balancers and the servers. Each server has a second 1 Gbps interface that is attached to a switch connected to the client.

The purpose of this setup is to mimic a production environment where the servers would have a dedicated network interface directly connected to the Internet. Clients download web pages, representing a total amount of 4 GB. We use the apache benchmark software [AB] to simulate 10 parallel clients. We use `netem` to simulate different delays and different packet loss ratios. To simplify the interpretation of the figures, we started by configuring the load balancer to send all requests to a single server. An evaluation with several servers is provided in Section 5.5.3.

Figure 5.10 shows the number of requests completed every second when the client downloads 4 GB using different web pages sizes. The evaluation shows that for small request sizes, Multipath TCP slightly underperforms TCP. This can be explained by the slightly higher cost of establishing Multipath TCP connections [RPB+12].

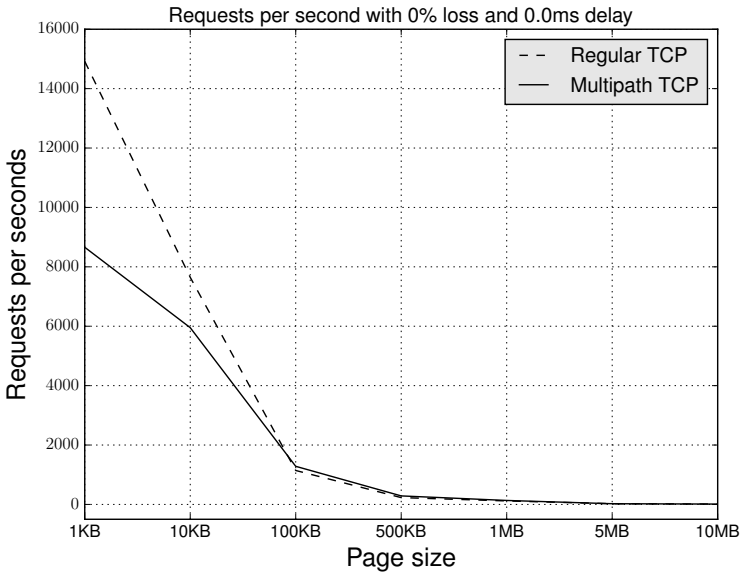


Figure 5.10: Number of requests per second without loss or delay.

For larger request sizes, starting at 100 KB, Multipath TCP and TCP both sustain the same number of requests per second, which is expected because TCP uses the 1 Gbps link connected to the load balancer, while Multipath TCP uses the 1 Gbps link connected directly to the server.

With this experiment, we want to test whether our solution is deployable in a production environment. In this chapter we argue that with our proposal it is no longer needed to use costly hardware to run a load balancer. To prove this point, the remaining measurements in this section have been run with the same scripts, but we changed the speed of the link between the client and the load balancer to 100 Mbps.

Figure 5.11 shows the transfer rate for the same experiment as the one shown in 5.10, but with a 100 Mbps link between the client and the load balancer. Again, for small request sizes, Multipath TCP slightly underperforms TCP. For larger requests such as 500 KB, Multipath TCP reaches a goodput of 942 Mbps. The higher Multipath TCP goodput is an illustration that Multipath TCP provides more than TCP. Indeed, shortly after the establishment of the initiation subflow, the client learns the address of the load balanced server and creates a second subflow via the 1 Gbps interface of the server. Multipath TCP then automatically uses the interface going directly to the server and achieves higher goodput than TCP.

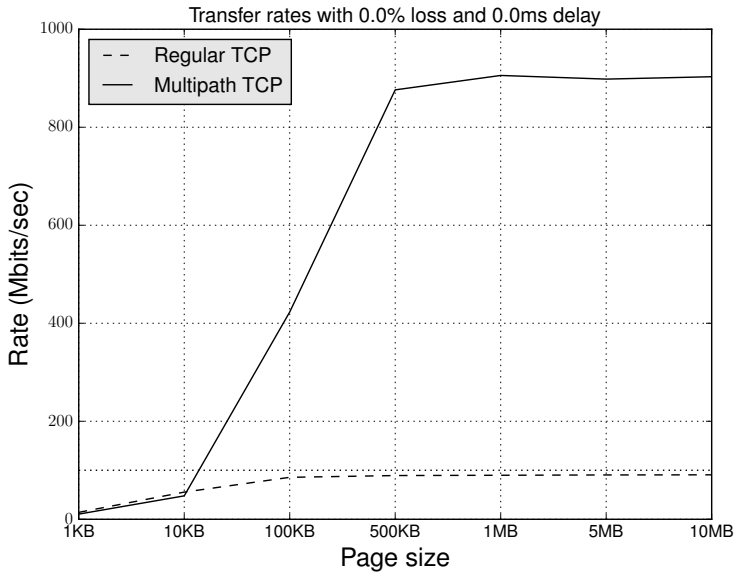


Figure 5.11: Transfer rates without loss or delay.

Impact of the delay

We evaluate in this section whether latency affects the performance of Multipath TCP behind load balancers.

For this experiment, we configure a delay of 20 ms on the link between the client and the load balancer. Figure 5.12 shows that Multipath TCP is still able to benefit from the 1 Gbps link. Unsurprisingly, the transfer rate for small web objects is lower than when there is no added latency. This is an expected and already documented [AGC+14b] behaviour of Multipath TCP. In our setup, Multipath TCP starts with an initial subflow that uses the 100 Mbps link. The client sends the HTTP GET over this subflow and it can only start the establishment of the second subflow after the reception of the acknowledgements for this initial data. The 20 ms added latency delays the establishment of the second subflow and thus lowers the total transfer rate.

By increasing the latency to 200ms, as shown in Figure 5.13 we see an important impact on both Multipath TCP and TCP. This high latency increases the time required for congestion control algorithm used on the subflows to ramp up.

Impact of packet losses

Packet losses are another factor that can influence the performance of TCP. Measurements over the global Internet have reported packet loss ratios of

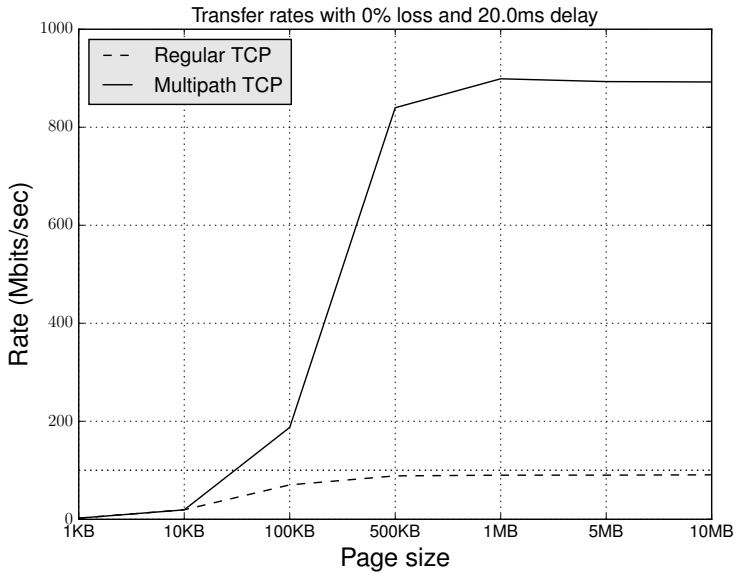


Figure 5.12: Transfer rates with no loss and 20ms delay.

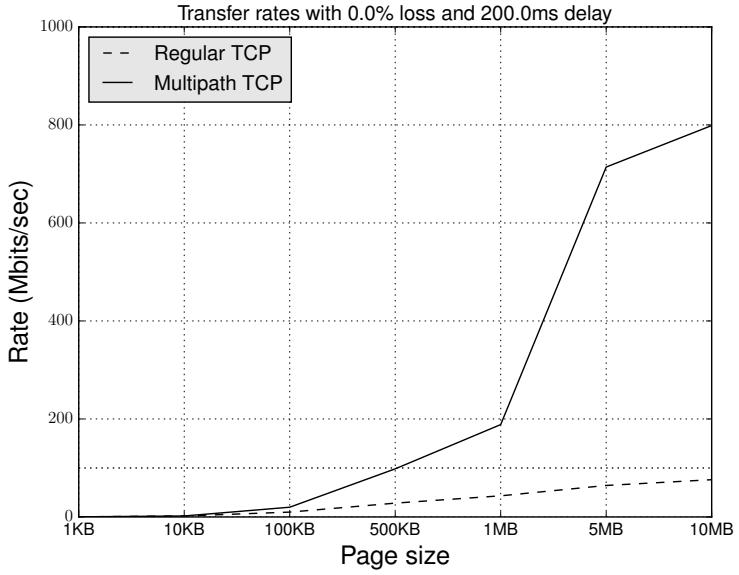


Figure 5.13: Transfer rates with no loss and 200ms delay.

roughly up to 1%. Our solution needs to cope with two different types of packet losses: *(i)* loss of a TCP packet and *(ii)* loss of a packet carrying the

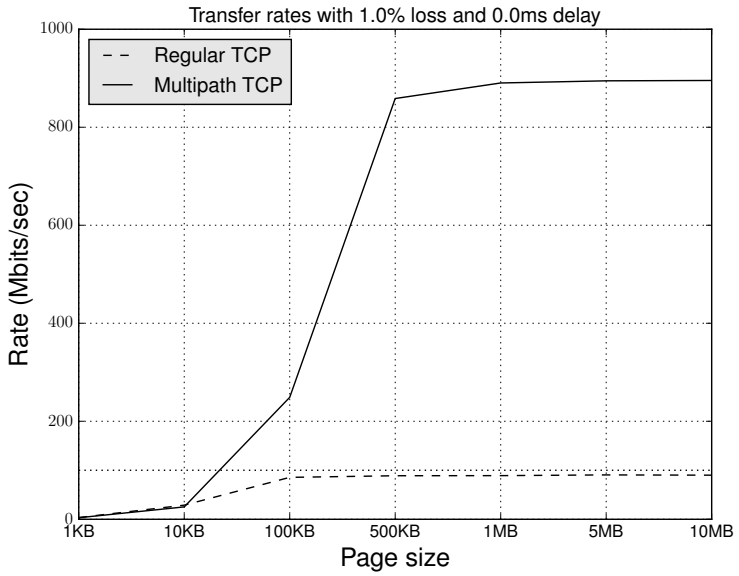


Figure 5.14: Transfer rates with 1% loss and no delay.

ADD_ADDR option that announces the physical address of the server. The standard retransmission and congestion control mechanisms used by TCP and Multipath TCP cope with the former type of packet losses. Our implementation copes with the latter by ensuring that the ADD_ADDR option is reliably delivered. If a packet carrying the ADD_ADDR option is lost, it is retransmitted later to ensure that the remote host has learned the new address. Figure 5.14 shows that when there is no added latency, the Multipath TCP throughput is not affected by packet losses. A closer look at the packet traces confirmed that a second subflow was created for all Multipath TCP connections.

Figure 5.15 shows that even when we combine loss and delay, the performance of Multipath TCP is not significantly affected compared to TCP. The important factor being the latency, this can be verified by comparing figures 5.15 and 5.13. The high latency playing only for the initial connection establishment, the performances are lower than without latency, but larger files sizes enable Multipath TCP to achieve a transfer rate of 803 Mbps where TCP achieves 16 Mbps. With these measurements, we demonstrated that with our modifications, Multipath TCP works in environments where Layer-4 load balancers are used. In this specific setup, we used Linux Virtual Server [LVS], but any Layer-4 load balancer can ben used with the same results. Our measurements show that with our modifications, the load balancer is no longer the bottleneck of the network since it is only used to put the client in relation with the server. With Multipath TCP, the load balancers no longer need to be

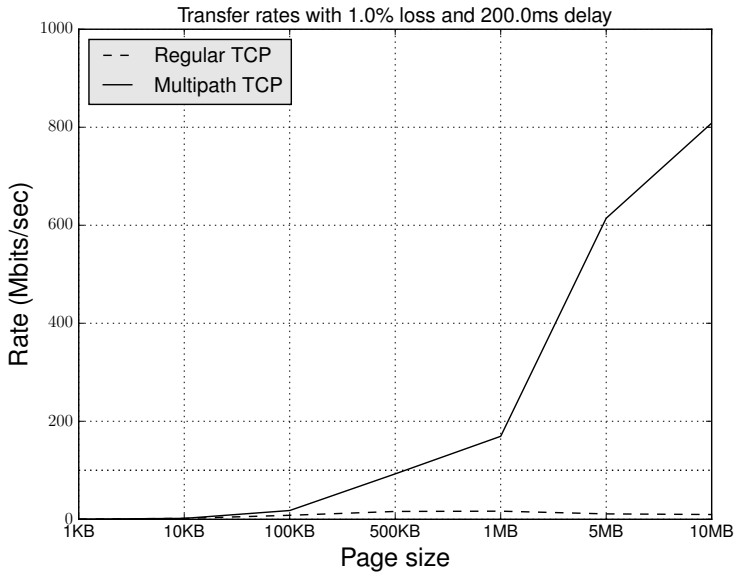


Figure 5.15: Transfer rates with 1% loss and 200ms delay.

expensive machines with a lot of power and network bandwidth as almost all of the traffic can be exchanged directly between the server and the client.

Direct Server Return (DSR) or Direct Return [LVSDR] improves performance in a download scenario, but does not bring benefits for upload scenarios, where most of the traffic is going from the client to the server like in storage scenarios. Our solution, however, fully works in both directions, allowing it to be used in more scenarios.

5.5.3 Anycast

A full evaluation of anycast would require a deployment in a larger network that was not possible given the number of servers in our lab. From an abstract viewpoint, an anycast deployment can be considered as a network that distributes the packets sent by clients to the closest server. If the network topology changes, some clients could be redirected to a different server. This change would affect TCP and this is the main reason why anycast TCP is difficult.

To evaluate the support of anycast services, we rely on the network shown in Figure 5.16. Each server has two addresses on its 1 Gbps interface: the anycast address and a unique address. Each server listens to the anycast address and they are configured to advertise their unique address with Multipath TCP and set the initial subflow as a backup subflow. These servers are behind a

router that uses Equal Cost MultiPath (ECMP) [Hop00] to distribute the load accross the servers. The client is connected to the router via a 10 Gbps link, while each server is connected to the router via a 1 Gbps link. As in the previous setup, the clients establish multiple HTTP connections, 300 in this case, and download files from these servers.

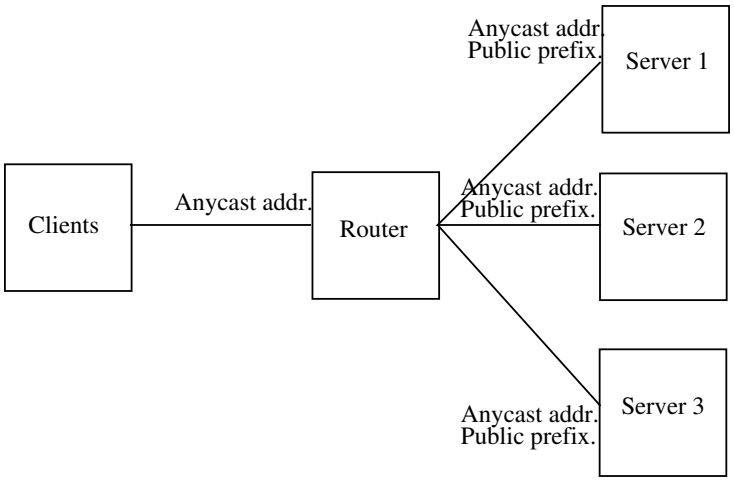


Figure 5.16: Anycast Evaluation setup.

To simulate network reconfigurations, every 10 seconds, we remove one of the server from the ECMP anycast pool during 5 seconds. After 5 seconds, this server is added again.

TCP Anycast

Figure 5.17 shows results obtained with TCP anycast. We run the measurement during 120 seconds. The client machine runs apache benchmark configured to retrieve very large (100 MBytes) files from one of the anycast servers. We use 300 parallel clients. The figure shows two different curves. The top curve plots the utilisation of the link between the client and the router. When all servers are part of the ECMP pool, the client downloads at 2800Mbps. However, when the router is reconfigured and one of the physical servers is removed from the pool to simulate a topology change in the network, the utilisation of the link drops to 1900 Mbps. This is expected since one server has left the ECMP pool. Unfortunately, a consequence of this network reconfiguration is that some packets towards the anycast address are redirected to a different server than before the topology change. Since this server does have state for the TCP connection, it sends a RST packet and the client needs to restart the entire download. A closer look at the bottom curve of Figure 5.17 reveals that servers also send RST packets when a new server is added to the

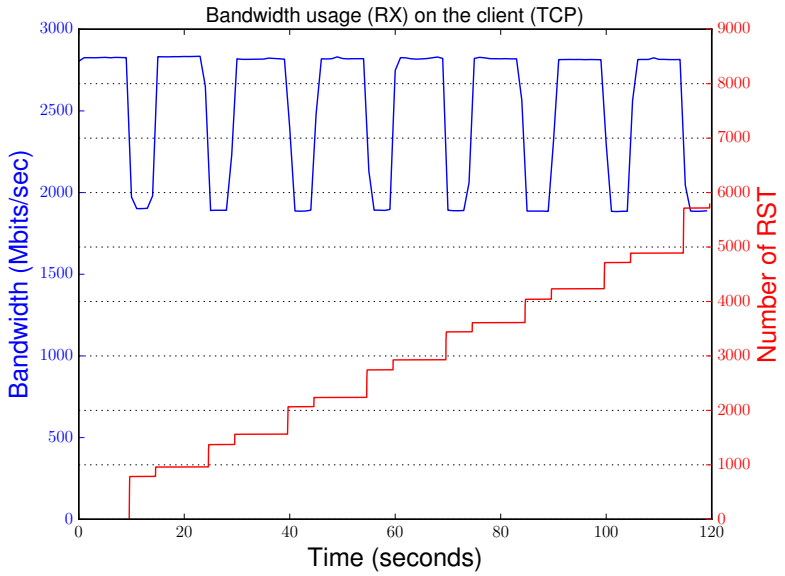


Figure 5.17: With TCP, many connections are reset and this affects the utilisation of the client-router link.

pool. We experimentally observe that more RST packets are generated when a server is removed from the ECMP anycast pool than when a server is added to the ECMP anycast pool. This is normal because when a server is removed from an ECMP anycast pool, all the TCP connections that were handled by this server are redirected. These RST packets explain why network operators do not want to deploy TCP anycast in entreprise networks.

We now perform exactly the same measurements with our extension to Multipath TCP. Figure 5.18 shows results that are completely different from those obtained with regular TCP. The first and most important result is that we do not observe any failure of established Multipath TCP connections during the 120 seconds of the experiment. Despite of the 16 topology changes that we simulated, no Multipath TCP connection failed. This is a very important result that confirms that Multipath TCP can be deployed to support anycast services which could bring an additional use case for Multipath TCP. The upper curve of Figure 5.18 reveals that the utilisation of the client-router bandwidth stays at 2.8 Gbps despite the network reconfigurations. When a server is removed from the ECMP anycast pool, the Multipath TCP connections that were handled by this server automatically switch to the second subflow as the initial subflow (towards the anycast address) is redirected to another physical server. This handover is seamless for the application and the connection continues. When a server is removed from the ECMP anycast

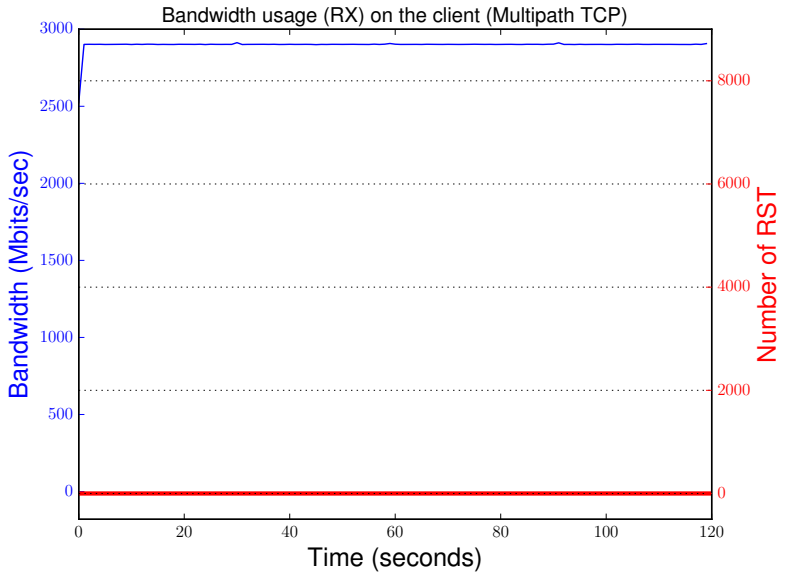


Figure 5.18: With Multipath TCP, no connection is reset when the network topology changes.

pool, the packets belonging to an initial subflow are redirected to another server that does not have state for this subflow. This server responds to those packets with RST packets that terminate this initial subflow. However, the second subflow, which is attached to the unique server address, is still up and the data transfer continues.

5.6 Security Considerations

Besides distributing the load among different servers, load balancers also shield the physical servers from the open Internet and can filter some of the packets sent to the physical servers depending on their configuration. By advertising the addresses of the physical servers, our solution exposes them more than existing stateless load balancers. If network operators are concerned about the advertisement of the addresses of the physical servers, there are several solutions that can be used to mitigate the security risks.

First, the physical servers only need to accept additional subflows. A security concerned network administrator can easily reject incoming `SYN` packets containing the `MP_CAPABLE` option to prohibit the establishment of new Multipath TCP connection that do not pass through the load balancer. Those filters can be installed on the physical servers or upstream firewalls. This could also be achieved by slightly modifying the Multipath TCP/TCP imple-

mentation to reject any SYN packets not containing the MP_JOIN option on specificed interfaces.

Another point that is worth to be discussed are the additional subflows that can be established by sending SYN packets with the MP_JOIN option towards the physical server that was selected by the load balancer. Multipath TCP [FRHB13] relies on two techniques to protect the servers from the establishment of fake subflows. First, the MP_JOIN option contains a 32-bits token that uniquely identifies the Multipath TCP connection. If an attacker wants to add a subflow to an existing Multipath TCP connection, they must guess the 32-bits token that identifies this connection. This is not sufficient since the establishment of the additional subflows is authenticated by using HMACs that are computed over 64 bits keys exchanged by the client and the server during the initial handshake. To successfully create an additional subflow, an attacker would need to guess this 64 bits key.

With the above solution, the server plays an active role in mitigating the attack since it needs to match the received SYN with the tokens that it has allocated and then compute the HMAC before sending the SYN+ACK. The computational cost of this HMAC could be a concern in the case of denial of service attacks. In our LAN, a single unique address has been assigned to each physical server. In IPv4 networks, this would be the expected deployment given the scarcity of IPv4 addresses. In IPv6 networks, many addresses are available. We could leverage the large IPv6 addressing space and allocate one /64 prefix to each physical server. The server would then announce this prefix to the network to which it is connected. When a new Multipath TCP connection arrives on the server, it assigns a unique IPv6 address from its /64 prefix to this specific connection. We propose to compute the low order 64 bits of this address as $hash(secret, token)$ where $hash$ is a fast hash function, $token$ the token associated to this connection and $secret$ a random number. The server then concatenates the output of this function to its /64 prefix and announces it to its client. This address is unique to this specific connection. If the client creates another subflow towards this server, it will send a SYN packet towards this address with the connection token inside the MP_JOIN option. A simple filter can then be used, either on the physical server or an upstream firewall to verify the validity of the SYN packet without requiring any state. This filter could be implemented as a set of eBPF rules similar to those described by Cloudflare in [Ber16]. Such eBPF rules can process packets at a higher rate than the Linux kernel and thus are very useful when mitigating denial of service attacks.

5.7 Conclusion

The deployment of Multipath TCP on servers has been hindered by the difficulty of supporting it on stateless load-balancers. In this chapter we have proposed a small modification to Multipath TCP that enables it to work behind any stateless load balancers. This modification has already been accepted by the IETF [FRH+19]. We have implemented our modifications to Multipath TCP in its reference implementation in the Linux kernel and have demonstrated its performance with measurements in the lab. An important benefit of our solution compared to existing deployments such as Direct Server Return, is that the load balancer can be placed off-path for long transfers. This is an important feature that could be very useful as the web transitions to the HTTP/2 protocol that will use longer connections than HTTP/1.x.

Our Multipath TCP extension is more generic that simply supporting load balancers in front of servers. It enables network operators to use anycast addresses for Multipath TCP services. This brings another use case for Multipath TCP in addition to the existing deployments that leverage fast failover or bandwidth aggregation.

Repeatability of the results

The measurement results described in this chapter were obtained with our modifications to the reference implementation of Multipath TCP in the Linux kernel. These modifications and the measurement scripts are available at `https://github.com/fduchene/ICNP2017` to enable other researchers to repeat our measurements and expand them.

Chapter 6

Conclusion

Over the last decades, the Internet has dramatically grown in size and in adoption. To meet the user's expectations in terms of reliability and performance, network engineers and operators designed and deployed several load balancing techniques like ECMP, Layer-4 load-balancers or Multipath TCP. In this thesis, we explored some of these techniques to identify potential weaknesses and possible new use cases with one idea in mind: finding simple solutions that scale in today's networks.

With this idea in mind, in Chapter 2 we reconsidered Multipath TCP's usage of subflows. First, we reconsidered the semantics of the backup subflow. We introduced the active backup scheduler that provides a trade-off between packet time delivery and the utilization of the backup interface by allowing the usage of a backup subflow for packets that have been delayed for a configurable amount of time. We implemented and evaluated this scheduler, showing that it keeps most of the traffic on the primary subflow for interactive applications such as web browsing. Then, we introduced the notion of trusted resource pooling with Multipath TCP enables specifying a level of trust per-interface. This level of trust allows protecting specific pieces of data like the keys contained in the `MP_CAPABLE` option or sensitive parts of unencrypted protocols. The evaluation of our implementation indicated that considering the trustiness of the interfaces has a very small impact on the raw performance of Multipath TCP.

In Chapter 3, we leveraged IPv6 Segment Routing's ability to enforce precise network paths to create an architecture that enables arbitrary in-network Virtual Functions, that can be applied on bytestreams and chained together. We explained that functions operating on the bytestream where opening the possibilities of new stateful uses cases like multimedia transcoding and Multipath TCP proxies. We also argued that one of the benefits of this solution is that the middleboxes are explicitly exposed, improving the manageability of the network. We implemented a proof-of-concept of our architec-

ture and evaluated it on several platforms. Our measurements indicated that our architecture is well suited to support explicit middleboxes that process bytestreams.

In Chapter 4, we proposed a new solution leveraging IPv6 Segment Routing and eBPF to efficiently steer transport flows. As it takes its inception in FlowBender, we first presented this solution that allows an efficient re-routing of flows using Equal Cost Multipath (ECMP) and Explicit Congestion Notification (ECN). Then, we presented our architecture leveraging IPv6 Segment Routing and eBPF's ability to access TCP's internal state to efficiently steer transport flows. We argue that our solution could be used for different use cases and is not limited to reacting to congestion events. With our implementation and simulated scenarios, we demonstrated that it is possible to improve the performance of transport protocols by dynamically selecting paths.

In Chapter 5, we took a different approach to make Multipath TCP friendlier with load balancers and anycast. We showed that despite its growing deployment on client devices, its deployment in datacenters is hindered by its incompatibility with current Layer-4 load-balancers. Other researchers tried to solve this problem by designing Multipath TCP-aware load-balancers. To respect Multipath TCP's design goal to be used in today's networks, we took the stance to solve this problem by modifying Multipath TCP. By slightly modifying the protocol, we designed an extension that enables it to work behind any stateless load-balancer. We also demonstrated that our Multipath TCP extension is more generic than simply supporting load balancers in front of servers. It enables network operators to use anycast addresses for Multipath TCP services, bringing another use case for Multipath TCP in addition to the existing deployments. We demonstrated its performance with measurements in the lab. Our extension is part of the next version of Multipath TCP that is finalized within the IETF.

Open problems

During this thesis, we opened several future research directions. First, while we demonstrated the benefits of using an expiration timer, finding the right value is difficult for the application. We believe that the application should be able to express its intent to the stack in a simple fashion (e.g., latency senstive connection or bulk transfer). The stack would then compute the adequate value of the expiration timer. In its deployment of Multipath TCP, Apple recently started to give the application this kind of simple configuration. Second, while our implementation of SRv6Pipes is generic and provides good performances, kernel bypass techniques such as DPDK [DPD] or userspace TCP stacks like mTCP [JWJ+14] could allow significant performance boosts and be considered for a future deployment. Finally, while eBPF's ini-

tial usage was limited to the Linux kernel, it has since broadened and researchers are now using eBPF in user space to dynamically extend protocols like QUIC [DCMP+19]. Meanwhile, eBPF's implementation in the Linux kernel is growing by the day, allowing researchers to extend protocols like TCP [TB19]. eBPF's broadening scope and the evolution of its implementation combined are opening new possibilities for our solution combining IPv6 Segment Routing and eBPF. These possibilities could be explored to address more use cases in the future.

Bibliography

[AB] AB, *Apache Bench*. Accessed: 2017-04-23.

[ABK+12] James W. Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat, *xomb: Extensible open middleboxes with commodity servers*, Proceedings of the eighth acm/ieee symposium on architectures for networking and communications systems, 2012, pp. 49–60.

[ABP13] N. AlFardan, D. Bernstein, and K. Paterson, *On the Security of RC4 in TLS*, USENIX Security (2013).

[ACF+17] Ahmed AbdelSalam, Francois Clad, Clarence Filsfils, Stefano Salsano, Giuseppe Siracusano, and Luca Veltri, *Implementation of virtual network function chaining through segment routing in a linux-based nfv infrastructure*, Ieee conference on network softwarization (netsoft), 2017July.

[ACO+06] Brice Augustin, Xavier Cuvellier, Benjamin Orgogozo, Fabien Viger, Timur Friedman, Matthieu Latapy, Clémence Magnien, and Renata Teixeira, *Avoiding traceroute anomalies with paris traceroute*, Proceedings of the 6th acm sigcomm conference on internet measurement, 2006, pp. 153–158.

[ADK13] Y. C. Hu A. Dixit P. Prakash and R. R. Kompella., *On the impact of packet spraying in data center networks*, 2013.

[AFLV08] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat, *A scalable, commodity data center network architecture*, Proceedings of the acm sigcomm 2008 conference on data communication, 2008, pp. 63–74.

[AGC+14a] B. Arzani, A. Gurney, Shuotian Cheng, R. Guerin, and Boon Thau Loo, *Impact of path characteristics and scheduling policies on MPTCP performance*, Advanced information networking and applications workshops (waina), 2014 28th international conference on, 2014May, pp. 743–748.

[AGC+14b] Behnaz Arzani, Alexander Gurney, Sitian Cheng, Roch Guerin, and Boon Thau Loo, *Deconstructing MPTCP performance*, Network protocols (icnp), 2014 ieee 22nd international conference on, 2014, pp. 269–274.

[AGM+10] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan, *Data Center TCP (DCTCP)*, Proceedings of the acm sigcomm 2010 conference, 2010, pp. 63–74.

[AL06] J. Abley and K. Lindqvist, *Operation of Anycast Services*, Internet Request for Comments, RFC Editor, RFC Editor, Fremont, CA, USA, 2006.

[amazontranscode] *Amazon Elastic Transcoder*. Accessed: 2018-04-05.

[AMK98] Elan Amir, Steven McCanne, and Randy Katz, *An active service framework and its application to real-time multimedia transcoding*, Acm sigcomm computer communication review, 1998, pp. 178–189.

[AVBD18] François Aubry, Stefano Vissicchio, Olivier Bonaventure, and Yves Deville, *Robustly disjoint paths with segment routing*, Proceedings of the 14th international conference on emerging networking experiments and technologies, 2018, pp. 204–216.

[BAM10] Theophilus Benson, Aditya Akella, and David A. Maltz, *Network traffic characteristics of data centers in the wild*, Proceedings of the 10th acm sigcomm conference on internet measurement, 2010, pp. 267–280.

[BAM13] Mehdi Bezahaf, Abdul Alim, and Laurent Mathy, *Flowos: A flow-based platform for middleboxes*, Proceedings of the 2013 workshop on hot topics in middleboxes and network function virtualization, 2013, pp. 19–24.

[Barracuda] *Barracuda Load Balancer ADC*. Accessed: 2017-04-23.

[BBG+19] Olivier Bonaventure, Mohamed Boucadair, Sri Gundavelli, SungHoon Seo, and Benjamin Hesmans, *0-RTT TCP Convert Protocol*, Technical Report draft-ietf-tcpm-converters-08, Internet Engineering Task Force, 2019. Work in Progress.

[BBJS14] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger, *TCP Extensions for High Performance*, Internet Request for Comments, RFC Editor, RFC Editor, Fremont, CA, USA, 2014.

[BCMHash] *Avoiding network polarization and increasing visibility in cloud networks using broadcom smart hash technology*. Accessed: 2013-07-17.

[BDG+14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker, *P4: Programming protocol-independent packet processors*, SIGCOMM Comput. Commun. Rev. **44** (July 2014), no. 3, 87–95.

[BDK18] BPF-DOC-KERNEL, *Linux kernel documentation - linux socket filtering aka berkeley packet filter (bpf)*, 2018. [Online; accessed 8 June 2018].

[Ber16] Gilberto Bertin, *Introducing the p0f BPF compiler*, 2016. `https://blog.cloudflare.com/introducing-the-p0f-bpf-compiler/`.

[BFM13] Luca Boccassi, Marwan M. Fayed, and Mahesh K. Marina, *Binder: A system to aggregate multiple internet gateways in community networks*, Proceedings of the 2013 acm mobicom workshop on lowest cost denominator networking for universal access, 2013, pp. 3–8.

[BH18] BPF-Helpers, *Bpf helpers - documentation*, 2018. [Online; accessed 8 June 2018].

[BHH+10] A. Bittau, M. Hamburg, M. Handley, D. Mazières, and D. Boneh, *The Case for Ubiquitous Transport-level Encryption*, Usenix security, 2010.

[bigip] *Release Note: BIG-IP LTM and TMOS 11.5.0*. Accessed: 2017-05-10.

[BJSE16] Deval Bhamare, Raj Jain, Mohammed Samaka, and Aiman Erbad, *A survey on service function chaining*, J. Netw. Comput. Appl. **75** (November 2016), no. C, 138–155.

[BL16] BPF-LWT, *bpf: Bpf for lightweight tunnel encapsulation*, 2016. [Online; accessed 8 June 2018].

[BL18] BPF-LWN, *Linux weekly news - a thorough introduction to ebpf*, 2018. [Online; accessed 9 June 2018].

[Bor16] Daniel Borkmann, *On getting tc classifier fully programmable with cls bpf*, Proceedings of netdev (2016).

[BP18] BPF-PERF, *perf examples*, 2018. [Online; accessed 8 June 2018].

[BPB11] Sébastien Barré, Christoph Paasch, and Olivier Bonaventure, *Multipath tcp: From theory to practice*, Ifip networking, valencia, 2011May.

[BPG+14] M. Bagnulo, C. Paasch, F. Gont, O. Bonaventure, and C. Raiciu, *Analysis of MPTCP residual threats and possible fixes*, Technical Report draft-ietf-mptcp-attacks-00, 2014.

[BS16] Olivier Bonaventure and SungHoon Seo, *Multipath TCP deployments*, IETF Journal **2016** (2016November). http://www.ietfjournal.org/multipath-tcp-deployments/.

[BX18] BPF-XDP, *Bpf and xdp reference guide*, 2018. [Online; accessed 8 June 2018].

[CB02] B. Carpenter and S. Brim, *Middleboxes: Taxonomy and Issues*, Internet Request for Comments, RFC Editor, RFC Editor, Fremont, CA, USA, 2002.

[CBHB16] Q. De Coninck, M. Baerts, B. Hesmans, and O. Bonaventure, *Observing real smartphone applications over Multipath TCP*, IEEE Communications Magazine **54** (2016March), no. 3, 88–93.

[CFP+07] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker, *Ethane: Taking control of the enterprise*, Sigcomm '07, 2007, pp. 1–12.

[CGKR10] Lorenzo Colitti, Steinar H. Gunderson, Erik Kline, and Tiziana Refice, *Evaluating ipv6 adoption in the internet*, Passive and active measurement, 2010, pp. 141–150.

[cisco] *Cisco cli command reference*. Accessed: 2013-07-17.

[CLG+13] Yung-Chih Chen, Yeon-sup Lim, Richard J. Gibbens, Erich M. Nahum, Ramin Khalili, and Don Towsley, *A measurement-based study of Multipath TCP performance over wireless networks*, Proceedings of the 2013 conference on internet measurement conference, 2013, pp. 455–468.

[CT14] Y. C. Chen and D. Towsley, *On bufferbloat and delay analysis of Multipath TCP in wireless networks*, Networking conference, 2014 ifip, 2014June, pp. 1–9.

[cznic] *CZ.NIC*. Accessed: 2017-08-28.

[Dal92] W. J. Dally, *Virtual-channel flow control*, IEEE Trans. Parallel Distrib. Syst. **3** (March 1992), no. 2, 194–205.

[Datanyze] *Datanyze - Load Balancers market share report*. Accessed: 2017-04-23.

[DCBHB16] Quentin De Coninck, Matthieu Baerts, Benjamin Hesmans, and Olivier Bonaventure, *A first analysis of Multipath TCP on smartphones*, International conference on passive and active network measurement, 2016, pp. 57–69.

[DCMP+19] Quentin De Coninck, François Michel, Maxime Piraux, Florentin Rochet, Thomas Given-Wilson, Axel Legay, Olivier Pereira, and Olivier Bonaventure, *Pluginizing QUIC*, 2019 acm sigcomm conference, 2019.

[DH98] S. Deering and R. Hinden, *Internet Protocol, Version 6 (IPv6) Specification*, Internet Request for Comments, RFC Editor, RFC Editor, Fremont, CA, USA, 1998. Updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045, 7112.

[DK89] Daniel M. Dias and Manoj Kumar, *Preventing congestion in multistage networks in the presence of hotspots*, Icpp (1)'89, 1989, pp. 9–13.

[dlOBC+11] A. de la Oliva, C. Bernardos, M. Calderon, T. Melia, and J. Zuniga, *IP Flow Mobility: Smart Traffic Offload for Future Wireless Networks*, IEEE Communications Magazine **49** (2011), no. 10, 124–132.

[DNSB14] Shuo Deng, Ravi Netravali, Anirudh Sivaraman, and Hari Balakrishnan, *WiFi, LTE, or both?: Measuring multi-homed wireless internet performance*, Proceedings of the 2014 conference on internet measurement conference, 2014, pp. 181–194.

[DPD] DPDK, *Dataplane Development Kit*. Accessed: 2019-07-18.

[ea15] Andrew Wood et al., *Pipe viewer*, URL http://www.ivarch.com/programs/pv.shtml (2015).

[ER13] J. Erman and K. Ramakrishnan, *Understanding the Super-sized traffic of the Super Bowl*, Acm imc, 2013.

[EV02] Cristian Estan and George Varghese, *New directions in traffic measurement and accounting*, Proceedings of the 2002 conference on applications, technologies, architectures, and protocols for computer communications, 2002, pp. 323–336.

[EYC+16] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein, *Maglev: A fast and reliable software network load balancer*, 13th usenix symposium on networked systems design and implementation (nsdi 16), 2016, pp. 523–535.

[Fab16] G. Fabregas, *Tr-349: Hybrid access broadband network architecture*, 2016. Broadband Forum.

[FAMB16] Simone Ferlin, Özgü Alay, Olivier Mehani, and Roksana Boreli, *BLEST: Blocking estimation-based MPTCP scheduler for heterogeneous networks*, Ifip networking 2016, May 2016.

[FBK+17] Adrienne Porter Felt, Richard Barnes, April King, Chris Palmer, Chris Bentzel, and Parisa Tabriz, *Measuring HTTPS adoption on the web*, 26th USENIX security symposium (USENIX security 17), 2017, pp. 1323–1338.

[FDP+19] Clarence Filsfils, Darren Dukes, Stefano Previdi, John Leddy, Satoru Matsushima, and daniel.voyer@bell.ca, *IPv6 Segment Routing Header (SRH)*, Technical Report draft-ietf-6man-segment-routing-header-21, Internet Engineering Task Force, 2019. Work in Progress.

[FGL+19] Clarence Filsfils, Pablo Camarillo Garvia, John Leddy, daniel.voyer@bell.ca, Satoru Matsushima, and Zhenbin Li, *SRv6 Network Programming*, Technical Report draft-filsfils-spring-srv6-network-programming-07, Internet Engineering Task Force, 2019. Work in Progress.

[FHG13] Xun Fan, John Heidemann, and Ramesh Govindan, *Evaluating anycast in the domain name system*, Infocom, 2013 proceedings ieee, 2013, pp. 1681–1689.

[FK97] Roy T Fielding and Gail Kaiser, *The Apache HTTP server project*, Internet Computing, IEEE **1** (1997), no. 4, 88–90.

[FLM+10] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin, *A First Look at Traffic on Smartphones*, Acm imc, 2010.

[FNP+15] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. Francois, *The Segment Routing Architecture*, 2015 IEEE Global Communications Conference (GLOBECOM), 2015Dec, pp. 1–6.

[For17] Mat Ford, *Landmark ipv6 report published: State of deployment 2017*, 2017. CircleID, `http://www.circleid.com/posts/20170606_landmark_ipv6_report_published_state_of_deployment_2017/`.

[FPG+18] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Bruno Decraene, Stephane Litkowski, and Rob Shakir, *Segment Routing Architecture*, Request for Comments, RFC Editor, 2018.

[FRHB13] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, *TCP Extensions for Multipath Operation with Multiple Addresses*, Internet Request for Comments, RFC Editor, RFC Editor, Fremont, CA, USA, 2013.

[FRH+19] Alan Ford, Costin Raiciu, Mark J. Handley, Olivier Bonaventure, and Christoph Paasch, *TCP Extensions for Multipath Operation with Multiple Addresses*, Technical Report draft-ietf-mptcp-rfc6824bis-18, Internet Engineering Task Force, 2019. Work in Progress.

[Fuk11] Kensuke Fukuda, *An analysis of longitudinal tcp passive measurements (short paper)* (Jordi Domingo-Pascual, Yuval Shavitt, and Steve Uhlig, eds.), Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[Gia13] Giovanni Giacobbi, *The GNU Netcat project*, URL http://netcat. sourceforge. net (2013).

[Goo13] D. Goodin, *Guerilla researcher created epic botnet to scan billions of ip addresses*, 2013. `http://goo.gl/G86ew`.

[Gro19] MAWI Working Group, *Packet traces from wide backbone*, 2019. `http://mawi.wide.ad.jp/mawi/samplepoint-G/2019/201906261400.html`.

[Haproxy] *HAProxy*. Accessed: 2017-04-23.

[HB16] Benjamin Hesmans and Olivier Bonaventure, *An Enhanced Socket API for Multipath TCP*, Proceedings of the 2016 applied networking research workshop, 2016, pp. 1–6.

[HDB+15] B. Hesmans, G. Detal, S. Barre, R. Bauduin, and O. Bonaventure, *SMAPP: Towards Smart Multipath TCP-enabled Applications*, Proceedings of the 11th acm conference on emerging networking experiments and technologies, 2015, pp. 28:1–28:7.

[HNR+11] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda, *Is it still possible to extend tcp?*, Proceedings of the 2011 acm sigcomm conference on internet measurement conference, 2011, pp. 181–194.

[Hop00] C. Hopps, *Analysis of an Equal-Cost Multi-Path Algorithm*, Internet Request for Comments, RFC Editor, RFC Editor, Fremont, CA, USA, 2000.

[howtopackage] *Creating OpenWRT packages*. Accessed: 2017-08-28.

[HP15] J. Halpern and C. Pignataro, *Service Function Chaining (SFC) Architecture*, Internet Request for Comments, RFC Editor, RFC Editor, Fremont, CA, USA, 2015.

[HQG+13] J. Huang, F. Qian, Y. Guo, Y. Zhou, and Q. Xu, *An in-depth Study of LTE: Effect of Network Protocol and Application Behavior on Performance*, Acm sigcomm, 2013.

[HRW14] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood, *Netvm: High performance and flexible networking using virtualization on commodity platforms*, Proceedings of the 11th usenix conference on networked systems design and implementation, 2014, pp. 445–458.

[HZH+16] Z. Hu, L. Zhu, J. Heidemann, A. Mankin, D. Wessels, and P. Hoffman, *Specification for DNS over Transport Layer Security (TLS)*, Internet Request for Comments, RFC Editor, RFC Editor, Fremont, CA, USA, 2016.

[ICA11] ICANN, *Available pool of unallocated ipv4 internet addresses now completely emptied*, 2011. `https://www.icann.org/en/system/files/press-materials/release-03feb11-en.pdf`.

[iperf] *iPerf - The ultimate speed test tool for TCP, UDP and SCTP*. Accessed: 2019-07-14.

[IT19] Jana Iyengar and Martin Thomson, *QUIC: A UDP-Based Multiplexed and Secure Transport*, Technical Report draft-ietf-quic-transport-22, Internet Engineering Task Force, 2019. Work in Progress.

[JB08] C. Jackson and A. Barth, *Forcehttps: Protecting High-Security Web Sites from Network Attacks*, Www, 2008.

[JB16] Kaustubh Joshi and Theophilus Benson, *Network function virtualization*, IEEE Internet Computing **20** (2016), no. 6, 7–9.

[JBB92] V. Jacobson, R. Braden, and D. Borman, *TCP Extensions for High Performance*, Internet Request for Comments, RFC Editor, RFC Editor, Fremont, CA, USA, 1992. Obsoleted by RFC 7323.

[JWJ+14] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park, *mtcp: a highly scalable user-level TCP stack for multicore systems*, 11th USENIX symposium on networked systems design and implementation (NSDI 14), 2014, pp. 489–502.

[KHP01] Christian Kreibich, Mark Handley, and V Paxson, *Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics*, Proc. usenix security symposium, 2001.

[KRV+15a] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, *Software-defined networking: A comprehensive survey*, Proceedings of the IEEE **103** (2015Jan), no. 1, 14–76.

[KRV+15b] Diego Kreutz, Fernando M. V. Ramos, Paulo Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig, *Software-Defined Networking: A Comprehensive Survey*, Proceedings of the IEEE **103** (2015), no. 1, 63.

[KVHD14] Abdul Kabbani, Balajee Vamanan, Jahangir Hasan, and Fabien Duchene, *Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks*, Proceedings of the 10th acm international on conference on emerging networking experiments and technologies, 2014, pp. 149–160.

[LB17] David Lebrun and Olivier Bonaventure, *Implementing IPv6 Segment Routing in the Linux Kernel*, Proceedings of the 2017 applied networking research workshop, 2017July.

[LC15] Yong Li and Min Chen, *Software-defined network function virtualization: A survey*, IEEE Access **3** (2015), 2542–2553.

[LD16] Simon Liénardy and Benoit Donnet, *Towards a Multipath TCP Aware Load Balancer*, Proceedings of the 2016 applied networking research workshop, 2016, pp. 13–15.

[Leb17] David Lebrun, *Reaping the benefits of ipv6 segment routing*, Ph.D. Thesis, 2017.

[Lei85] Charles E. Leiserson, *Fat-trees: universal networks for hardware-efficient supercomputing*, IEEE Trans. Comput. **34** (October 1985), no. 10, 892–901.

[LHM10] Bob Lantz, Brandon Heller, and Nick McKeown, *A network in a laptop: rapid prototyping for software-defined networks*, Proceedings of the 9th acm sigcomm workshop on hot topics in networks, 2010, pp. 19.

[LHZ+17] N. Leymann, C. Heidemann, M. Zhang, B. Sarikaya, and M. Cullen, *Huawei's GRE Tunnel Bonding Protocol*, Internet Request for Comments, RFC Editor, RFC Editor, Fremont, CA, USA, 2017.

[LJC+18] David Lebrun, Mathieu Jadin, François Clad, Clarence Filsfils, and Olivier Bonaventure, *Software resolved networks: Rethinking enterprise networks with ipv6 segment routing*, Sosr'18: Symposium on sdn research, 2018.

[LLV18] LLVM, *The llvm compiler infrastructure - project website*, 2018. [Online; accessed 9 June 2018].

[LLY+13] K. Lee, J. Lee, Y. Yi, I. Rhee, and S. Chong, *Mobile Data Offloading: How Much Can WiFi Deliver?*, IEEE/ACM Transactions on Networking **21** (2013), no. 2, 536–550.

[LVSDR] *Direct return in Linux Virtual Server*. Accessed: 2017-05-12.

[LVS] LVS, *Linux Virtual Server*. Accessed: 2019-07-18.

[MHFB15] Olivier Mehani, Ralph Holz, Simone Ferlin, and Roksana Boreli, *An Early Look at Multipath TCP Deployment in the Wild*, Proceedings of the 6th international workshop on hot topics in planet-scale measurement, 2015, pp. 7–12.

[mininet] *An Instant Virtual Network on your Laptop (or other PC)*. Accessed: 2019-07-14.

[MJ93] Steven McCanne and Van Jacobson, *The bsd packet filter: A new architecture for user-level packet capture.*, Usenix winter, 1993.

[MMFR96] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, *TCP Selective Acknowledgment Options*, Internet Request for Comments, RFC Editor, RFC Editor, Fremont, CA, USA, 1996.

[MSM13] J. Milliken, V. Selis, and A. Marshall, *Detection and analysis of the Chameleon WiFi access point virus*, EURASIP Journal on Information Security **2013** (2013), no. 1, 1–14.

[M+08] Nick McKeown et al., *Openflow: Enabling innovation in campus networks*, SIGCOMM Comput. Commun. Rev. **38** (March 2008), no. 2, 69–74.

[MZK+17] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu, *Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics*, Sigcomm '17, 2017, pp. 15–28.

[Netscaler] *Citrix Netscaler*. Accessed: 2017-04-23.

[NG14] E. Nordmark and I. Gashinsky, *Neighbor Unreachability Detection Is Too Impatient*, Internet Request for Comments, RFC Editor, RFC Editor, Fremont, CA, USA, 2014.

[NG16] Mehdi Nikkhah and Roch Guérin, *Migrating the internet to ipv6: an exploration of the when and why*, IEEE/ACM Transactions on Networking **24** (2016), no. 4, 2291–2304.

[NGI] NGINX, *Nginx*. Accessed: 2017-04-23.

[ns3] *NS-3 network simulator*. Accessed: 2013-07-17.

[OL15] Bong-Hwan Oh and Jaiyong Lee, *Constraint-based Proactive Scheduling for MPTCP in Wireless Networks*, Comput. Netw. **91** (November 2015), no. C, 548–563.

[openwrt] *OpenWRT*. Accessed: 2017-08-28.

[OR16] Vladimir Olteanu and Costin Raiciu, *Datacenter scale load balancing for multipath transport*, Proceedings of the 2016 workshop on hot topics in middleboxes and network function virtualization, 2016, pp. 20–25.

[Paa14] Christoph Paasch, *Improving Multipath TCP*, Ph.D. Thesis, 2014.

[PB12] C. Paasch and O. Bonaventure, *Securing the MultiPath TCP handshake with external keys*, Technical Report draft-paasch-mptcp-ssl-00, 2012.

[PB+] Christoph Paasch, Sebastien Barre, et al., *Multipath TCP in the Linux Kernel*. available from http://www.multipath-tcp.org.

[PDD+12] C. Paasch, G. Detal, F. Duchene, C. Raiciu, and O. Bonaventure, *Exploring Mobile/WiFi Handover with Multipath TCP*, Acm sigcomm cellnet workshop, 2012, pp. 31–36.

[perf] *perf: Linux profiling with performance counters*. Accessed: 2018-03-29.

[PFAB14] Christoph Paasch, Simone Ferlin, Ozgu Alay, and Olivier Bonaventure, *Experimental evaluation of Multipath TCP schedulers*, Proceedings of the 2014 acm sigcomm workshop on capacity sharing workshop, 2014, pp. 27–32.

[PGF15] Christoph Paasch, Greg Greenway, and Alan Ford, *Multipath TCP behind Layer-4 loadbalancers*, Technical Report draft-paasch-mptcp-loadbalancer-00, Internet Engineering Task Force, 2015. Work in Progress.

[PJ13] Rahul Potharaju and Navendu Jain, *Demystifying the dark side of the middle: a field study of middlebox failures in datacenters*, Proceedings of the 2013 conference on internet measurement conference, 2013, pp. 9–22.

[PMM93] Craig Partridge, Trevor Mendez, and Walter Milliken, *Host anycasting service*, 1993.

[Pos81a] J. Postel, *Internet Protocol*, Internet Request for Comments, RFC Editor, RFC Editor, Fremont, CA, USA, 1981. Updated by RFCs 1349, 2474, 6864.

[Pos81b] Jon Postel, *Transmission Control Protocol*, Internet Request for Comments, RFC Editor, RFC Editor, Fremont, CA, USA, 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[qdisc-hhf] *net-qdisc-hhf: Heavy-Hitter Filter (HHF) qdisc*. Accessed: 2019-07-14.

[QEP17] P. Quinn, U. Elzur, and C. Pignataro, *Network Service Header (NSH)*, 2017. Internet draft, draft-ietf-sfc-nsh-28.

[RBP+11] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, *Improving Datacenter Performance and Robustness with Multipath TCP*, Acm sigcomm 2011, 2011.

[Res01] E. Rescorla, *SSL and TLS: Designing and Building Secure Systems*, Addison Welsey, 2001.

[RFB01] K. Ramakrishnan, S. Floyd, and D. Black, *The Addition of Explicit Congestion Notification (ECN) to IP*, Internet Request for Comments, RFC Editor, RFC Editor, Fremont, CA, USA, 2001. Updated by RFCs 4301, 6040.

[RNBH11] Costin Raiciu, Dragos Niculescu, Marcelo Bagnulo, and Mark James Handley, *Opportunistic mobility with Multipath TCP*, Proceedings of the sixth international workshop on mobiarch, 2011, pp. 7–12.

[RPB+12] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley, *How hard can it be? Designing and implementing a deployable Multipath TCP*, Proceedings of the 9th usenix conference on networked systems design and implementation, 2012, pp. 29–29.

[RSS] *RSS: Receive Side Scaling*. Accessed: 2018-08-28.

[San12] C. Sankaran, *Data Offloading Techniques in 3GPP Rel-10 Networks: A Tutorial*, IEEE Communications Magazine **50** (2012), no. 6, 46–53.

[Sch10] Michael Scharf, *Multi-Connection TCP (MCTCP) Transport*, Technical Report draft-scharf-mptcp-mctcp-01, Internet Engineering Task Force, 2010. Work in Progress.

[SCP+16] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford, *Pisces: A programmable, protocol-independent software switch*, Sigcomm '16, 2016, pp. 525–538.

[SE01] P. Srisuresh and K. Egevang, *Traditional IP Network Address Translator (Traditional NAT)*, Internet Request for Comments, RFC Editor, RFC Editor, Fremont, CA, USA, 2001.

[SEKF13] Philipp S. Schmidt, Theresa Enghardt, Ramin Khalili, and Anja Feldmann, *Socket Intents: Leveraging Application Awareness for Multi-access Connectivity*, Proceedings of the ninth acm conference on emerging networking experiments and technologies, 2013, pp. 295–300.

[Seo] SungHoon Seo, *KT's GiGA LTE: Commercial Mobile MPTCP Proxy service launch*. https://www.ietf.org/proceedings/93/slides/slides-93-mptcp-3.pdf.

[SHS+12] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar, *Making middleboxes someone else's problem: network processing as a cloud service*, ACM SIGCOMM Computer Communication Review **42** (2012), no. 4, 13–24.

[Soc11] The Internet Society, *State of ipv6 deployment 2018*, 2011. `https://www.internetsociety.org/resources/2018/state-of-ipv6-deployment-2018/`.

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark, *End-to-end arguments in system design*, ACM Trans. Comput. Syst. **2** (November 1984), no. 4, 277–288.

[STJ03] J. R. Santos, Y. Turner, and G. Janakiraman, *End-to-end congestion control for infiniband*, Ieee infocom 2003. twenty-second annual joint conference of the ieee computer and communications societies (ieee cat. no.03ch37428), 2003March, pp. 1123–1133 vol.2.

[TB19] Viet-Hoang Tran and Olivier Bonaventure, *Beyond socket options: making the linux tcp stack truly extensible*, The ifip networking 2019 conference, 2019May.

[TMB10] J. Touch, A. Mankin, and R. Bonica, *The TCP Authentication Option*, Internet Request for Comments, RFC Editor, RFC Editor, Fremont, CA, USA, 2010.

[turris] *Turris Omnia*. Accessed: 2017-08-28.

[UTE17] Motomu Utsumi, Hajime Tazaki, and Hiroshi Esaki, *`/dev/stdpkt`: A service chaining architecture with pipelined operating system instances in a unix shell*, Aintec '17: Asian internet engineering conference, 2017November 20–22.

[VB10] C.A. Visaggio and L.C. Blasio, *Session Management Vulnerabilities in Today's Web*, IEEE Security & Privacy **8** (2010), no. 5, 48–56.

[WBKW14] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall, *How speedy is SPDY?*, 11th usenix symposium on networked systems design and implementation (nsdi 14), 2014, pp. 387–399.

[WHB08] D. Wischik, M. Handley, and M. Bagnulo, *The resource pooling principle*, SIGCOMM Comput. Commun. Rev. **38** (September 2008), no. 5.

[wrk] *wrk - a HTTP benchmarking tool*. Accessed: 2017-12-31.

[XLS05] Jun Xin, Chia-Wen Lin, and Ming-Ting Sun, *Digital video transcoding*, Proceedings of the IEEE **93** (2005), no. 1, 84–97.

[ZDM+12] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz, *Detail: reducing the flow completion time tail in datacenter networks*, Proceedings of the acm sigcomm 2012 conference on applications, technologies, architectures, and protocols for computer communication, 2012, pp. 139–150.

[ZHH+15] Liang Zhu, Zi Hu, John Heidemann, Duane Wessels, Allison Mankin, and Nikita Somaiya, *Connection-oriented DNS to improve privacy and security*, Security and privacy (sp), 2015 ieee symposium on, 2015, pp. 171–186.