Evaluating and Improving the Deployability of Multipath TCP

Gregory Detal

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

March 11, 2014

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

Thesis Committee:

Pr. Gildas **Avoine**	UCL/ICTEAM, Belgium
Pr. Olivier **Bonaventure** (Advisor)	UCL/ICTEAM, Belgium
Pr. Marco **Canini**	UCL/ICTEAM, Belgium
Pr. Pierre **Francois**	IMDEA Networks, Spain
Pr. Laurent **Mathy**	Université de Liège, Belgium
Pr. Charles **Pecheur** (Chair)	UCL/ICTEAM, Belgium
Pr. Costin **Raiciu**	Universitatea Politehnica din Bucuresti, Romania

Evaluating and Improving the Deployability of Multipath TCP
by Gregory Detal

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

This work was partially supported by the European-funded CHANGE project as well as the ALAWN WIST project from the Walloon Region.

Preamble

The concept of Internet started in the early 1970s with the deployment of the first research network ARPANet [Abb99]. At that time, various protocols have been developed to interconnect different networks [Cla88]. The concept of a worldwide interconnection of networks, called the Internet, started with the standardization of the *Internet protocol suite* (TCP/IP). In the late 1980s, commercial *Internet Service Providers* (ISPs) emerged, leading to the commercial Internet that we know today. This public Internet has revolutionized our society by impacting our culture and commerce, providing an easy access to information and near-instant communication [Cas01].

The Internet never stopped evolving. Recently, smartphones and tablets have emerged. Compared to classical hosts that use a single network interface, a major innovation brought by these devices is that they have several network interfaces. Since these are wireless interfaces, mobility becomes possible. However, even if the Internet evolved, the design principles of the TCP/IP protocol suite still apply. The suite was designed to provide a mean to transfer data from one end host to the other. At that time, the networks were single paths. There is thus a mismatch between the network that is multipath and the protocols that are still single path.

This mismatch often causes some frustration for Internet users. Indeed, a user that has a smartphone with both WiFi and 3G connectivity could expect to be able to maintain its connections (e.g., to listen to a web radio) while moving. It is however not the case today, even if a WiFi and/or a 3G connectivity is always available. Once the smartphone loses its WiFi connectivity all ongoing TCP connections stall.

With the emergence of smartphones and tablets, mobile data usage also increases. This growth of data traffic and smart devices forces telco companies to find a better solution to deal with mobile multihomed hosts. One of the solutions is to offload data traffic on WiFi networks [abi09]. Indeed, WiFi hotspots can be built economically and expanded incrementally. For example, AT&T deployed free Hotspots in Time Square NYC because its 3G

data network was overloaded [Vos10]. WiFi networks are very fragmented and mainly present in airports, hotels and city centers [abi09]. The number of open WiFi networks is growing thanks to Fon. Fon [fon13] is an initiative to allow end users to share their WiFi access point with other members of the initiative. Fon claims to have the largest WiFi network in the world, with over 12 million hotspots as of July 2013. Offloading 3G connections to WiFi impacts user experience similarly to moving, i.e., losing WiFi connectivity. All connections established through 3G will stall when moving to WiFi.

Several proposals have been proposed during the last decade to provide resilience while in presence of multiple paths [NB09, MN06, Ste07]. Unfortunately, none of these have been widely deployed. One of these solutions is the *Stream Control Transmission Protocol* (SCTP) [Ste07]. SCTP defines a new transport protocol that supports multihoming to provide transparent failover. However, designed in 2000, SCTP is not yet widely deployed even if it brings the desired features to improve the end-user experience. The reason is quite simple. The End-to-End principle, which states that application-specific functions ought to reside in the end hosts rather than in intermediate nodes, does not apply anymore on today's Internet. Other devices than routers and switches are used to provide connectivity between a client and a server [SHS+12]. There exist middleboxes which modify/drop packets and inspect the network layer or above. A well known middlebox is the *Network Address Translator* (NAT) that is present in almost every set-top box. SCTP failed to be deployed due to the lack of support by most middleboxes. Indeed, most middleboxes only support TCP, UDP and sometimes ICMP.

Multipath TCP (MPTCP) [FRHB13] is a more recent proposal that is standardized at the IETF. The designers of MPTCP solved the deployability issue of SCTP by taking middleboxes into account [RPB+12]. They designed the protocol as an extension to TCP as to appear transparent to the network and so to middleboxes. MPTCP main objective is to use all the network resources available [WHB08]. On top of providing resiliency, MPTCP increases the throughput of a single connection by using multiple paths simultaneously.

In this thesis we focus on MPTCP. MPTCP is rather new and there does not exist validation that it works in today's network. More specifically, the main goal of this thesis is to evaluate the deployability of MPTCP in various networks from mobile to data center networks as well as to improve the protocol. The main contributions of this thesis are:

- We implement the support of mobility inside the reference Linux implementation of MPTCP and show that end hosts can benefit from it over today's networks.

- We evaluate and improve the deployability of MPTCP. More precisely, we provide a tool to detect middleboxes on the Internet without requiring server cooperation. This allows us to evaluate the deployability of MPTCP on the current Internet.
- We propose to deploy MPTCP-TCP converters on the Internet to allow smartphones to use MPTCP when contacting legacy servers. This could encourage smartphones manufacturers to deploy MPTCP as it allows end hosts to benefit from it. We have implemented an efficient prototype and provide a in-depth evaluation of the latter.
- We evaluate the usage of MPTCP inside data center networks and provide a new way of performing load balancing as to enable multipath protocols to more easily use all available paths.

Roadmap

This thesis is decomposed in five chapters. **Chapter 1** provides the background. It presents the motivation for MPTCP as well as an in-depth overview of the protocol.

In **Chapter 2**, we evaluate the benefits of using MPTCP in a real environment. More precisely, we evaluate the impact of performing a WiFi/3G handover on the performances of various applications. We show that MPTCP could indeed provide an important benefit for smartphone users by increasing the overall end user experience.

Chapter 3 presents `tracebox` a software tool that allows to detect middlebox interference on the Internet. This chapter provides an assessment of `tracebox` as well as an evaluation of middleboxes interference on TCP extensions.

MPTCP as every new protocol/extension suffers from the *chicken-and-egg* problem. There is a no incentive for one host to support the protocol if the other one does not support it. We tackle this issue in **Chapter 4** by deploying MiMBox, a MPTCP-TCP converter, that enables end users to benefit from MPTCP with servers that does not yet support it.

In **Chapter 5**, we propose and evaluate a new load-balancing algorithm for data center networks while allowing MPTCP to easily benefit from this network diversity in order to increase its performance.

Finally, we conclude this thesis and provide perspective for further work in **Chapter 6**.

Bibliographic notes

Most of the work presented in this thesis appeared in conferences proceedings and journals. The list of related publications is shown hereafter:

- S. van der Linden, G. Detal and O. Bonaventure. *Revisiting Next-hop Selection in Multipath Networks*. In Proceedings of ACM SIGCOMM Poster Session, 2011. Toronto, Canada.
- C. Paasch, G. Detal, F. Duchêne, C. Raiciu and O. Bonaventure. *Exploring Mobile/WiFi Handover with Multipath TCP*. In Proceedings of the ACM SIGCOMM workshop on Cellular Networks (Cellnet'12), 2012. Heslinki, Finland.
- G. Detal, C. Paasch, S. van der Linden, P. Mérindol, G. Avoine and O. Bonaventure. *Revisiting Flow-Based Load Balancing: Stateless Path Selection in Data Center Networks*. Computer Networks, 57(5):1204-1216, April 2013.
- G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel and B. Donnet. *Revealing Middlebox Interference with Tracebox*. In Proceedings of the 2013 ACM SIGCOMM conference on Internet Measurement Conference, October 2013. Barcelona, Spain.
- G. Detal, C. Paasch and O. Bonaventure. *Multipath in the Middle(Box)*. In Proceedings of the ACM CoNEXT workshop HotMiddlebox'13, December 2013. Santa Barbara, USA.
- B. Hesmans, F. Duchêne, C. Paasch, G. Detal, O. Bonaventure. *Are TCP extensions middlebox-proof?*. In Proceedings of the ACM CoNEXT workshop HotMiddlebox'13, December 2013. Santa Barbara, USA.

Several publications appeared in journals and workshops but were out of the scope of this thesis:

- G. Detal, D. Leroy and O. Bonaventure. *An Adaptive Three-Party Accounting Protocol*. In Proceedings of the CoNEXT Student Workshop '09, pages 3-4, December 2009. Rome, Italy.
- D. Leroy, G. Detal, J. Cathalo, M. Manulis, F. Koeune and O. Bonaventure. *SWISH: Secure WiFi Sharing*. Computer Networks, Special Issue on "Network Convergence", 55(7):1614-1630, May 2011.

Acknowledgments

The fulfillment of this thesis would not have been possible without the support of many people to whom I am greatly indebted. First, I want to thank my wife Delphine for the continuous source of encouragement and my daughter Lily for the smile she puts on my face every day. I am grateful as well to my parents Jean-Louis and Christiane for the support they provided throughout my life.

I am also grateful to Olivier, my advisor, who guided me through this thesis. Even if it was not always easy, he defined most of the major directions of my work and redirected me when required. I am also thankful to him for the lot of time he spent and will send to promote with me the results of my research.

I am grateful to all my coauthors, Damien Leroy, Julien Cathalo, Mark Manulis, Francois Koeune, Simon van der Linden, Christoph Paasch, Fabien Duchêne, Costin Raiciu, Pascal Mérindol, Gildas Avoine, Benjamin Hesmans, Yves Vanaubel and Benoit Donnet. Some of their writing appears in this document.

I am also really thankful to the members of my thesis committee, Gildas Avoine, Marco Canini, Pierre Francois, Laurent Mathy, Charles Pecheur and Costin Raiciu. Thank you all for having taken time to review my thesis, and for the interesting discussions during the private defense. You gave me a lot of interesting feedback and valuable comments for improving my manuscript.

My former and current colleagues have helped me, more or less directly and technically, to complete this thesis. The main ones are Damien Leroy, Damien Saucez and Christoph Paasch. Many thanks to them but also to the other members of the team alongside whom I have worked for short or long periods of time: Bruno Quoitin, Sébastien Barré, Benoit Donnet, Pierre Francois, Laurent Vanbever, Stefano Vissicchio, Juan Antonio

Cordero, Sébastien Dawans, Fabien Duchêne, David Lebrun, Benjamin Hesmans, Simon van der Linden, Virginie Van den Schrieck. I also wish to thanks the whole INGI department, academic and administrative, for all their help during those years at the UCL.

Gregory Detal
March 11, 2014

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Background

The Internet is part of everyone's life. The services provided by the Internet are numerous. It enables an easy access to information, social contacts, gaming, shopping, etc. Today, more than 35% of the world population has an Internet access at home while in developed countries this number reaches up to 80% [int13]. The services are provided thanks to the various components of the Internet architecture that are decoupled from each other and that together provide a way to move IP packets from point A to B. One of these components is the data plane. It is the actual mechanism used to transmit a packet over the transit paths.

The data plane of the Internet was designed more than 30 years ago. At that time the Internet was composed of 3 types of devices: end hosts, switches and routers. Figure 1.1 shows the Internet protocol suite (TCP/IP) model that characterizes and standardizes the internal functions of a communication system by partitioning it into abstraction layers. The model groups similar communication functions into one of five logical layers. A layer serves the layer above it and is served by the layer below it. Layers 1 and 2 are related to access mediums (WiFi, LTE, 3G, Ethernet, etc.), i.e., to the communication between directly connected nodes. Layer 3 is called the network layer and provides an unreliable delivery of data packets between end hosts. The transport layer provides end-to-end communication services. Among others, it provides convenient services such as connection-oriented data stream support, reliability and flow control. It relies heavily on the network layer to provide reliable data transfer for the upper layers. The last layer corresponds to the application layer and provides a mean for the users' applications to interact with the network stack. The Internet is dominated by the *Internet Protocol* (IP) at the network layer, the *Transport Control Protocol* (TCP) at the transport layer [San13] and the *HyperText Transfer Protocol* (HTTP) [San13] at the application layer.

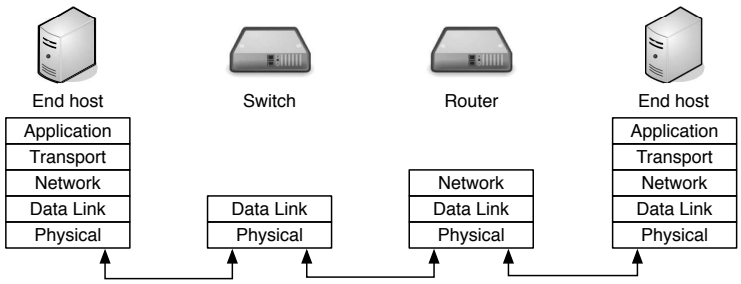


Figure 1.1: The Internet data plane as it was when the TCP/IP stack was designed.

1.1 The Internet evolution

The Internet has reshaped itself over the past decade. This began with the need for failover and load spreading across networks. As demands on the Internet grew, load balancing techniques have been developed to provide a more reliable network and improve performances. Load balancing is used in different types of networks from ISP networks to data center networks. Load balancing allows network operators to maximize the throughput, reduce congestion as well as to achieve redundant connectivity in their network. Load balancing can be applied at different layers of the Internet protocol suite from the data link layer to the transport layer. The most frequently used technique in the network layer is called *Equal Cost Multi-Path* (ECMP) [Hop00]. A common implementation uses a hash function over the 5-tuple (IP source and destination, protocol and, port source and destination) to select a next hop amongst the equally-good next hops available. ECMP provides that multiple connections between the same end hosts do not follow a single path but are distributed within the network.

More recently, the edge of the network undergoes a high evolution toward multihomed end hosts. Indeed, the last couple of years have seen a huge boom in mobile devices usage. The number of smartphones and tablets has exploded [CS13a, Rus12]. As of 2011, there were 1 billion smartphones users which represented 1/5 of the mobile phone users. This number is continuing to grow and the number of smart devices is expected to reach 10 billion in 2016, larger that the number of people living on the planet at that time. The global mobile data traffic grew 70% in 2012 and is expected to grow more in the coming years due to the deployment of new technologies such as 4G And LTE [CS13a]. Smartphone users desire to be 'always connected', the smart devices therefore often contains multiple Internet interfaces such as WiFi and 3G/LTE/4G.

Different resources are available at both the network and the end-host level. Unfortunately, the protocols still in-use today were designed with single-path networks in mind. TCP connections are intrinsically linked to IP. When a TCP connection is established, it is bound to the IP addresses of the two communicating hosts. If one of these addresses changes, for whatever reason, the connection fails.

This mismatch between today's multipath networks and TCP's single-path design creates tangible problems. For instance, if a smartphone loses its WiFi signal, the TCP connections associated with it stall. There is no way to migrate them to other working interfaces, such as 3G. This makes mobility a frustrating experience for users. Modern data centers are another example: many paths are available between two endpoints, and multipath routing randomly picks one path for a particular TCP connection [Hop00]. This can cause collisions where multiple flows get placed on the same link, hurting throughput.

1.2 Providing multipath support

Several proposals have tried to use this network diversity as to use the multiple paths available. These solutions mainly operate at the network layer or above. We discuss in the rest of this section various solutions at the different layers.

1.2.1 At the network layer

Shim6 [NB09, BRB11], as its name suggests, is a shim layer between the network and the transport layer. Shim6 works over IPv6 and hides the changes of IP addresses to the transport layer. Shim6 defines failure detection and locator pair exploration functions which allows hosts to detect and recover from failures. To achieve these features, specific Shim6 probes or hints from the above layer are used. As Shim6 consists of a modification of the network stack, existing applications automatically benefit from its functionality.

While bringing the desired multipath failover feature, Shim6 and other similar solutions such as Mobile IP [Per10] and MIPv6 [PJA11] are not used or widely deployed. Experiences have shown that performing mobility, i.e., handover between two network, at the network layer impacts the performances of the above layers. Changing the path used can introduce rapid changes in available capacity and delay. Hiding the changes of IP addresses to TCP can cause it to collapse. For example, moving from a non congested path to a highly congested paths will cause lot of losses that TCP will have

to deal with. Most likely, TCP will have to go into the slow start phase taking time to recover from the path change.

1.2.2 At the application layer

The application layer could provide a mobility mechanism. For example, after a new address is obtained, the session layer may simply initiate new transport connections to replace the existing ones. However, there are two main disadvantages to provide mobility at this layer. First, as the connections are restarted state must be maintained somewhere. For real-time streaming like connections this does not poses problem, the connection will restart at the expected time. For other cases it is not as easy, e.g., for a file download, if the connection is restarted the download will have to restart from the beginning increasing the overhead in the network. HTTP/1.1 [FGM+99] provides a way to define a recovery mechanism by indicating the amount of data already received in the header (the HTTP range field). While this could improve the mobility this feature only applies for static content and to this specific protocol.

1.2.3 At the transport layer

Researchers have shown that the best place to deal with mobility is at the transport layer [Edd04, WHB08]. Indeed, this layer has a better view of the paths congestion and thus can more efficiently decide which path to use and how to react to paths changes.

The *Stream Control Transmission Protocol* (SCTP) [Ste07] was designed to support multihoming and failover. The *Concurrent Multipath Transfer* (SCTP-CMT) extension of SCTP [IAS06] allows end hosts to use multiple paths at the same time improving therefore the overall performance. Another extension, mSCTP [SXT+07, KCL04], has been proposed to dynamically update a peer's address-list and thus allow handover from one interface to the other. Unfortunately SCTP has seen very little deployment due to the presence of middleboxes. Moreover, SCTP defines a different socket API than TCP. Applications must therefore be updated in order to benefit from SCTP.

At design time, the Internet was supposed to interconnect end hosts through routers and switches (see Figure 1.1) that only operate at the network layer or below. The TCP connection at the transport layer was supposed to be handed only by the end hosts. This is often referred to as the *End-to-End principle* [SRC84]. Unfortunately, the End-to-End principle does not apply anymore. Indeed, there exist devices on top of routers and

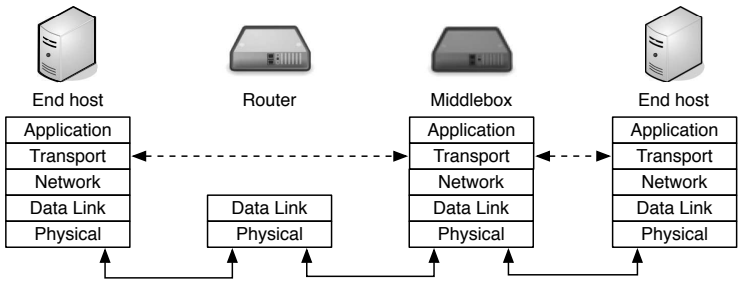


Figure 1.2: The Internet data plane as it is today.

switches within the network that provide services other than simple packet delivery. These devices interfere with the TCP connections by modifying fields of the header or the byte stream. The *Network Address Translator* (NAT), that is present in almost every home's set-top box, is one example. The NAT translates local IP addresses to public ones, as well as port numbers. By modifying the transport layer (and sometimes above), the NAT breaks the End-to-End principle.

This type of devices is often called *middleboxes*. A middlebox, defined as "*any intermediary box performing functions apart from normal, standard functions of an IP router on the data path between a source host and destination host*" in [CB02], manipulates packets for purposes other than simple forwarding. Figure 1.2 shows an up to date version of Figure 1.1 where these devices are present. Middleboxes are present everywhere. In addition to home networks, enterprise networks also contain a vast diversity of middleboxes from firewalls to proxies [SHS+12]. Although these middleboxes are supposed to be transparent to the end user, experience shows that they have a negative impact on the evolvability of the TCP/IP protocol suite [HNR+11]. After more than ten years of existence, SCTP is still not widely deployed, partially because many firewalls and NAT still consider SCTP as an unknown protocol and block the corresponding packets. SCTP is however used in specific application in controlled networks such as signaling in VoIP data networks [3GP12].

A more recent solution, *Multipath TCP* (MPTCP) [FRHB13] has been developed with the same idea in mind. However, compared to SCTP, its designers considered middleboxes while designing the protocol. To ensure deployability, they have defined the protocol as an extension to TCP. This is probably the most ambitious recent extension to TCP. It raised a lot of interest from researchers as well as from industry [Bon13]. The next section provides an in-depth overview of the protocol.

1.3 Multipath TCP

Multipath TCP is a new extension to TCP standardized by the *Internet Engineering Task Force* (IETF) [FRHB13]. MPTCP was designed to use all the available network resources [WHB08]. In order to be deployable, the authors took the following objectives into account while designing the protocol:

Do not require application changes
MPTCP does not require to rewrite all applications. For this, MPTCP is implemented in the TCP/IP stack of the operating system kernel and provides the same socket API as TCP.

Transparent to the network
The protocol must operate on today's Internet, it should deal with middleboxes. For this MPTCP uses regular TCP connections, called *subflows*. It uses TCP options to carry signaling and multiplexes the data stream over the subflows belonging to the same MPTCP connection. MPTCP maps each data to its own specific sequence number called *Data Sequence Number* (DSN). The DSN is present in all segments allowing the peer to reorder the data received over different subflows.

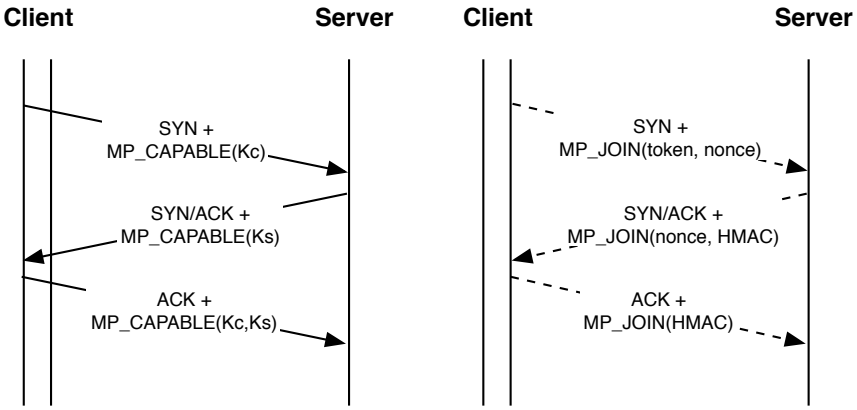
Backward compatible
To ensure that connectivity can be established with end hosts that do not support MPTCP, a fallback mechanism ensures that in this case regular TCP is used.

Moreover, the fallback mechanism is also used to deal with middleboxes. As MPTCP distributes data over multiple paths, the different paths might contain a middlebox that performs some modification to the segments. In this case, MPTCP fallbacks to regular TCP meaning that MPTCP signaling stops being used.

The operation of MPTCP can be decomposed in four different parts: *(i)* connection establishment, *(ii)* subflow establishment, *(iii)* data exchange and *(iv)* connection termination or subflow removal.

1.3.1 Connection establishment

The connection establishment is similar to TCP. A TCP connection starts with a three-way handshake. The client sends a SYN segment to the destination address of the server and to the destination port on which the server's service is listening to. The server replies with a SYN+ACK segment acknowledging the client's SYN. The client then acknowledges the



(a) Initial subflow. (b) Additional subflows.

Figure 1.3: Multipath TCP initial and additional subflows establishment.

SYN+ACK and the connection is established for both peers. The negotiation of TCP extensions is carried out during this handshake. For the *selective acknowledgement* (SACK) [MMFR96], the client sends the *SACK Permitted* TCP option in the SYN. If the server supports this extension and wants to use it for the new connection, it replies with the same option in the SYN+ACK.

In MPTCP, the peers negotiate the usage of the extension with the MP_CAPABLE option. Figure 1.3(a) shows the exchange in detail. The MP_CAPABLE options also contains keying material that are used to authenticate additional subflows. On top of being used for authentication purposes, the keys are also used to derive a unique identifier for this MPTCP connection. This unique identifier (32 bits) is called the *token* and is generated for both the client and the server. When sending its key the peer must ensure that the 32-bit token is unique on its side. The 64-bit *Initial Data Sequence Number* (IDSN) is also extracted from the keys. The algorithm used to derive the token and IDSN from the keys is exchanged in the option. As of today, a single mechanism based on HMAC-SHA1 is available. More information can be found in [FRHB13]. The keys and the token are further used during the additional subflow establishment in order to identify and authenticate the connection.

The security of MPTCP has been designed to be no worse than TCP. In TCP, an attacker that resides on the path between the client and the server can perform a *Man-in-the-Middle* attack where she can hijack or inject data in the connection [HH99]. As MPTCP uses regular TCP connections as subflows it is also impacted by the same problem. MPTCP cannot therefore

stop this Man-in-the-Middle attack. MPTCP designers chose therefore to exchange the keying material during the initial three-way handshake.

Compared to other extension, MPTCP also sends the MP_CAPABLE option in the third ACK where both keys are present. There are two reasons for this. First, it allows the server to answer a connection attempt without creating state in a similar fashion as SYN COOKIES [Ber96]. This prevents Denial-of-Service attacks with SYN flooding. Second, it allows to detect whether a middlebox has removed the option on the server-to-client path. In that case, both peers fallback silently to TCP because the path is not transparent to MPTCP.

1.3.2 Subflow establishment

Similarly to the initial subflow establishment, additional subflows are created through a three-way handshake. This is to overcome existing middleboxes such as firewalls that require a three-way handshake before any data transfer using a given 5-tuple. Figure 1.3(b) shows the control messages exchanged during the subflow establishment. MPTCP uses the MP_JOIN option to notify that the connection establishment corresponds to a new subflow. The mechanisms used in the establishment is twofold: identify the connection and authenticate both peers. To identify the connection the client sends the server's token in the SYN as well as a random number that will be used by the authentication mechanism. The authentication mechanism's goal is to prove that both hosts know a shared secret: the keys exchanged during the handshake of the initial subflow. To authenticate themselves, each peer sends a HMAC-SHA1 of each other's nonce using the concatenation of both keys.

Compared to regular TCP, the client cannot send data inside the third ACK. The subflow is considered in a `PRE-ESTABLISHED` state until a fourth ACK has been received by the client. This ensures that the third ACK is sent reliably and that the path can be used. This ACK has to be retransmitted. The connection state is changed to `ESTABLISHED` once an ACK has been received by the client, allowing to resume normal operation.

To establish subflows the peers have to be aware of each other's addresses. MPTCP associates one ID to each address. By default each peer knows the addresses used during the initial subflow establishment. From this they can establish new subflows to the peer address using their additional addresses. They identify their new address by setting the ID field in the MP_JOIN option.

Another way of notifying addresses to the other peer is to use the ADD_ADDRESS option. This option carries the address ID and an IPv4 or

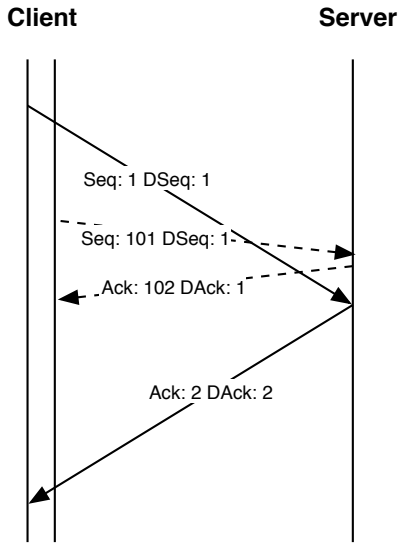


Figure 1.4: Multipath TCP data transfer.

IPv6 address that can be used by the peer to establish additional subflows. This notably allows MPTCP to establish IPv4 and IPv6 subflows within the same MPTCP connection. This option is sent in ACKs and is thus sent unreliably.

MPTCP allows both hosts to establish subflows. However, it is expected that only the active opener, i.e., the client, will initiate additional subflows. Indeed, nowadays, NAT devices are ubiquitous. It is therefore highly probable that a client is behind a NAT. NAT maintains state and is likely to drop incoming SYN. MPTCP also deals with ADD_ADDRESS containing locally routable addresses. When receiving the MP_JOIN for the same ID with a different address as announced in the ADD_ADDRESS, the peer has to update its ID to address mapping in order to ensure the correct usage of the address for the rest of the connection.

1.3.3 Data transfer

A naive implementation of MPTCP could distribute data amongst the available subflows and use regular TCP sequence numbers to allow the end hosts to reorder and recover the data stream. This however would cause holes in the TCP sequence number space of each subflow. Unfortunately middleboxes might maintain TCP state (e.g., a stateful firewall) for each subflow and check the validity of each TCP segment without knowing MPTCP. This might force the middlebox to close the connection. To ensure

a reliable transfer of data, MPTCP uses the Data Sequence Numbers as a second sequence number space. Each subflow maintains its own sequence space for each segment and writes this TCP sequence number as for a regular TCP connection. On top of this, each segment contains the *Data Sequence Signal* (DSS) MPTCP option that carries the DSN. The DSS option also contains the *Data ACK* (DACK). The DSS acts as a mapping on a piece of the data stream. For this it also contains information about the data mapped by carrying the subflow sequence number (relative to the initial sequence number) and the number of bytes covered by the mapping. The DSS option also contains a checksum computed on the data covered. This allows the MPTCP peers to detect middleboxes that modify the data stream in order to fallback to regular TCP. More information on fallback can be found in [FRHB13, HDP+13].

Figure 1.4 shows a simple data exchange between a client and a server using two subflows. In this simple example, the client sends one byte on each subflow. The first subflow has a higher delay than the other one which causes the server to receive the first byte of data after the second one. In the figure, `Seq` and `Ack` represent respectively the subflow sequence and acknowledgement numbers. The `DSeq` and `DAck` represent respectively the data sequence and acknowledgment numbers at the MPTCP-level. When receiving the second byte on the second subflow, the server acts as TCP at the subflow level, i.e., it acknowledges the data received since the data is in order regarding to the subflow sequence number. The byte is stored in an *Out-of-Order* queue at the MTPCP-level, waiting for data to fill the holes, before sending the data to the application. The server therefore replies with a DACK of 1 indicating that it is waiting for DSN 1. When the first byte arrives, it fills the hole and can thus be delivered to the application. The server replies a regular ACK on the first subflow containing the DACK 2. This confirms that it has correctly received the two first bytes.

Figure 1.4 is used as an illustration. The reality is more complex. There are retransmissions at the subflow level but also at the MPTCP level. The latter are called reinjections. Reinjection happens when a loss is detected. In this case, it can be more efficient to retransmit the data on another subflow especially if the loss happened on a high delay path. MPTCP can easily reinject data by remapping the data sequence numbers to new subflow sequence numbers. Each subflow must however still retransmit the original segments in order to deal with middleboxes.

Scheduler

One important feature of MPTCP is its scheduler. It distributes data amongst the active subflows. Various scheduling algorithms can be used.

The reference implementation of MPTCP [BPB11] in the Linux kernel tries to first send data on the subflow that has the lowest *Round-Trip Time* (RTT). It estimates the subflows' RTT thanks to the use of the Timestamp extension of TCP [JBB92]. The scheduler will order the subflows based on this delay measurement from the lowest to the highest one. The best subflow is preferred to send data. Data will only be sent over the second best subflow if the congestion window of the best subflow is full. The same applies for the other subflows. Using the RTT allows MPTCP to better react to network changes such as a smartphone moving away from a WiFi access point.

The scheduler depends on the OS on both hosts, one OS might implement a different scheduler than the other. MPTCP can notify the peer of its preferences in terms of subflow usage. The protocol can give a binary information, either the subflow should be used as backup. The notification is carried inside MP_JOIN that contains a backup bit or within the MP_PRIO option to change the priority during the lifetime of a subflow. Once a peer receives such priority modification it should not send data on backup subflow if there still exist non-backup ones.

Congestion control

The congestion control algorithm is an important feature of TCP [Jac88]. It dynamically adapts the throughput in response to changing network conditions. Each peer maintains a congestion window which limits the rate at which the packets are sent. The congestion control has two phases. First, the slow start phase happens at the beginning of the connection. At this moment the congestion level of the network is unknown and the sender tries to identify it. During this phase, the sender increases its congestion window exponentially until the first loss. The sender then enters the second phase: the congestion avoidance phase. In this phase, the congestion window grows linearly and is halved when a loss occurs. The congestion control algorithm allows to ensure fairness amongst independent connections that shares the same bottleneck. Each of these connections should converge to the same average value of the congestion window and thus the same rate.

Using MPTCP and a regular congestion control algorithm, would cause the different subflows to use an unfair share of the available capacity. If multiple subflows belonging to the same MPTCP connection have the same bottleneck links they will gain more throughput. The congestion control of MPTCP should ensure that if multiple subflows share the same bottleneck, an MPTCP connection would not get more throughput than regular TCP. Several congestion control algorithms have been presented in [RHW11, KGP+12]. The algorithm used by default in the Linux implementation is

the *Linked-Increases Algorithm* (LIA) [RHW11]. In this algorithm each subflow maintain its own congestion window which is halved when loss occur like in regular TCP. The increase phase is different, MPTCP increases the congestion window on the less congested subflow so that they increase proportionally more than congested ones. The global increase is dynamically chosen so that MPTCP remains fair. More details can be found in [RHW11, KGP+12].

1.3.4 Connection termination

There are two mechanisms to consider: *(i)* a subflow termination due to the loss of reachability via an IP address or *(ii)* the MPTCP connection termination after the call to `close()` by an application.

Loss of IP address

When a smartphone is mobile, it often moves from one WiFi access point to another and therefore looses IP Addresses during the handovers. If we consider a smartphone that has two IP addresses: a WiFi one and a 3G one. The smartphone has established two subflows using both addresses. When the smartphone looses the WiFi connectivity, it loses the associated address and so the subflow on its side. In TCP, the server has to wait for several minutes before considering the connection as over. MPTCP contains a mechanism to avoid keeping unused subflows. For this, the Remove_Addr option is used to signal to the peer the loss of an IP address. The option carries the ID associated with the address. In the above scenario, once the loss is discovered, the smartphone sends a Remove_Addr option over the 3G subflow to notify that the WiFi address has been lost. The server then reacts by sending a TCP RST on all subflows that use the address and reinject all data that were sent on the subflows and not yet acknowledged.

It should be noted that an MPTCP connection can survive without established subflows. Consider a smartphone that has a single WiFi connectivity and moves from one access point to the other. Its established MPTCP connection will stall between the access point but thanks to the Mp_Join option it will be able to further resume by establishing new subflows.

Termination due to `close()`

TCP uses the FIN flag to notify the peer that no more data will be further sent. As for the SYN, the FIN flag must be acknowledged by the peer. A double exchange of FIN guarantees a complete termination of the

connection. In MPTCP, relying on the successful exchange of FIN on all subflows does not guarantee a successful termination. Indeed, all subflows might not be operational when a peer wants to close the connection. To guarantee the successful termination, MPTCP uses the *Data FIN* signal that is carried inside the DSS option.

The Data FIN works similarly to the TCP FIN. It is sent to signal the last sequence number of the data. The Data FIN occupies one byte in the data sequence number space and is acknowledged by a Data ACK. It can be sent on any subflow.

When the application closes the connection, the Data FIN is sent on one or several subflows in addition to sending TCP FIN for each subflow. If the Data FIN is lost the reinjection mechanism of MPTCP will ensure that it is retransmitted on another subflow and that it is correctly received by the peer before removing the connection state.

1.4 Conclusion

In this chapter, we have presented Multipath TCP that is the basis for the thesis. We have explained the motivation for this protocol which is considered the next new major TCP extension. MPTCP brings many advantages to the end user such as resource pooling and mobility. The later advantage is of the highest importance for smartphone users that represent a large portion of the today's end users. Smartphones have often different interfaces but lack today of mobility. Smartphone users are often highly mobile as they always carry their phone and move from WiFi connected environment (e.g., home, office) to 3G connected ones. During handover, all their connections are broken and most of the time have to be restarted manually which is a major burden. With MPTCP, these users can always maintain their connections.

Chapter 2

Exploring the WiFi to 3G vertical handover

2.1 Introduction

A key concern on smartphones is the usage of the mobile data interface. Indeed, 3G often costs more money and is more energy-hungry than WiFi. 3G and the more recent LTE both suffer from what is called the *tail energy*. This causes the devices to continue consuming energy during a few seconds after sending data [AVRG+13]. 3G networks have also difficulties in supporting the growing bandwidth consumed by recent smartphones and tablets. Industry experts expect that these bandwidth requirements will continue to grow in the future and the deployment of new cellular technologies such as LTE may not be sufficient to sustain the demand [CS13a]. For these three reasons, smartphone users as well as network operators prefer to use private or public WiFi over 3G. As we have seen in Chapter 1, switching from one network to the other will make all ongoing TCP connections stall. Users will often have to manually restart their applications when moving out of the range of a wireless network which therefore impacts user experience. MPTCP can play an important role in solving this issue as it allows the connections to survive the loss of IP addresses.

Section 1.3 described in detail how MPTCP handles mobility, i.e., by dynamically add and remove IP addresses during the lifetime of a connection. In this chapter, we explore a little bit further this feature by looking at how MPTCP performances can be impacted when performing a WiFi to 3G handover. This chapter's goal is not to assess that MPTCP indeed brings advantages and improves the end-user experience. Previous work by Raiciu et al. have described why MPTCP is a promising solution for mobility [RNBH11] and used simulations to assess the possible benefits.

The aim of this chapter is to study experimentally how MPTCP reacts in a vertical handover scenario to give hints on how MPTCP could behave in other environments. A vertical handover usually refers to device changing the type of connectivity. In this chapter we look at how MPTCP behaves in the case of WiFi to 3G vertical handover.

This chapter is organized as follows. Section 2.2 describes how one can configure the MPTCP stack to perform handover. Section 2.3 evaluates the impact of the handover performance perceived by the end user. Section 2.4 presents a modification to the MPTCP stack to improve handovers efficiency. Finally, Sections 2.5 and 2.6 respectively compares MPTCP with the state-of-the-art mobility techniques and concludes the chapter.

2.2 Supporting vertical handovers

WiFi generally provide higher speeds and consumes less energy, while cellular technologies generally provide more ubiquitous coverage. In a smartphone environment, a user might want to use a WiFi connection whenever one is available, and to *fall over* to a cellular connection when the WiFi is unavailable. This is defined as a *vertical handover*. A *horizontal handover* differs from the fact that the same technology is used and the handover involves changes at the datalink layer instead of the network layer.

A mobile node should be able to adapt its protocol stack to its user's requirements when dealing with a potential vertical handover. From the user's viewpoint, there are three important factors to be considered. The first is the performance of the data transfer. Some users will probably prefer the fastest possible data transfer. The second factor is the battery lifetime. Some users will probably trade performance for longer battery lifetime. The third factor is traffic pricing. Some 3G networks bill in function of the number of transmitted bits or packets. Some users will favor cheaper networks such as WiFi hotspots, possibly for some applications.

The first and third factors are easily understandable. In this section, we first explore why data connections can drastically reduce the battery lifetime and then we present three potential modes of configuration of MPTCP.

2.2.1 The 3G state machine

Figure 2.1 depicts the *Radio Resource Control* (RRC) state machine for 3G as defined by 3GPP [3GP]. The state machine determines when a device can send or receive data. The main purpose is to control how many

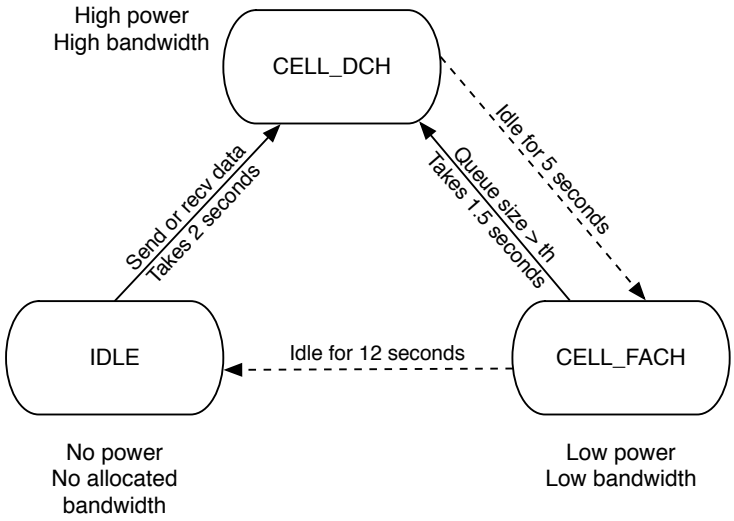


Figure 2.1: The 3G state machine.

devices can access the radio network at a give time as to limit network interferences.

We start describing the figure starting from the bottom left and continue proceeding clockwise. The first state the 3G device is when it starts is the IDLE state. In this state the radio is off and so the device does not consumes any energy. When data is to be sent, the device has to be promoted to the *Dedicated Channel* (CELL_DCH) state. Due to radio signaling, there is a 2 second delay before actually reaching the state. In this state the device is allocated a dedicated downlink and uplink channel and consumes around 800mW depending on the device [QWG+11]. When the device stops sending or receiving data and is idle for around 5 seconds, it is demoted to the *Forward Access Channel* (CELL_FACH) state. This demotion delay, called the *tail energy*, causes the device to continue consuming high power while the device is idle. In the CELL_FACH state, the device is not allocated a dedicated channel anymore but can still transmit data but at a lower rate. A device typically consumes around 400mW in this state. The device is further demoted to the IDLE state if there is still no data exchange or promoted back to the CELL_DCH state otherwise. Note that the transition timers and energy consumption numbers vary depending on the device and the carrier used.

Compared to 3G, WiFi consumes less energy thanks to its *Power Saving Mode* (PSM) feature [ACGP08]. PSM allows a device to switch its WiFi radio on and off several times a second to save battery. WiFi therefore does not suffers from the tail energy problem.

2.2.2 Mobility modes

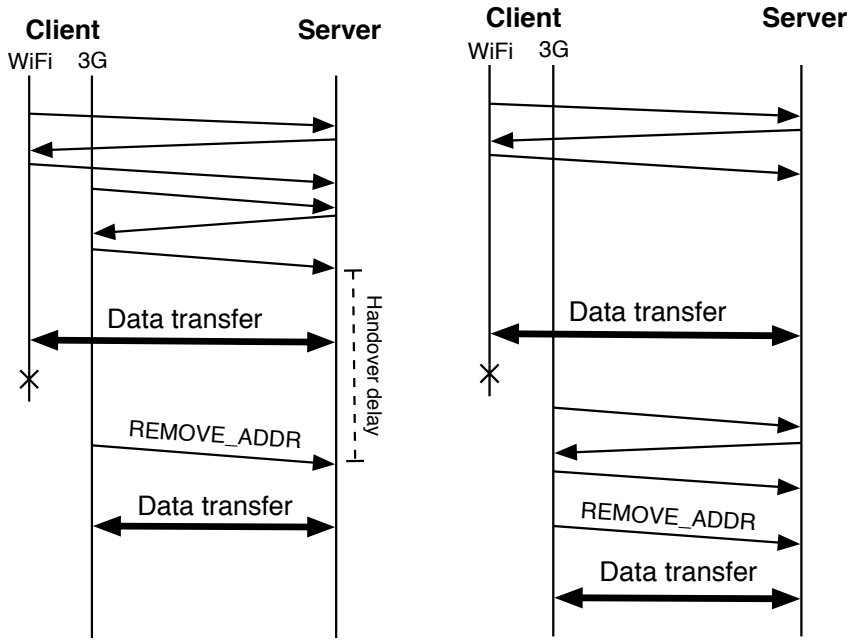
An MPTCP implementation should take these factors into account. In our current implementation, we have identified and implemented three modes of operation for MPTCP that correspond to most user needs.

***Full-MPTCP* mode**
The *Full-MPTCP* mode refers to the regular MPTCP operations where all subflows are used, i.e., where a full mesh of TCP subflows among the clients and the servers addresses is created. This mode is mostly intended for users who want to obtain the best data transfer rates.

***Backup* mode**
MPTCP opens TCP subflows over all interfaces as the *Full-MPTCP* mode, but uses only a subset of these to transport data segments. The MPTCP protocol allows an endpoint to signal to its peer that it should avoid sending data on a specific subflow by sending the MP_PRIO option on this subflow (see Section 1.3.3 for more details). This enables MPTCP to optimize for cost or battery lifetime by preferably sending data over the cheaper interfaces. It makes sense for MPTCP to prefer the WiFi interface when available and only use the 3G interface when there is no WiFi connectivity.

***Single-Path* mode**
The *Single-Path* mode allows a similar behavior as the *Backup* mode, except that at any moment only a single subflow is established and used for each MPTCP connection. When the interface goes down, the *Single-Path* mode establishes a new TCP subflow over another interface. This is possible thanks to the *break-before-make* design of MPTCP. It allows a short period of time during which no subflow is active. MPTCP is able to recover from this interruption by establishing a new TCP subflow and continue the data transmission without disturbing the application. Compared to the *Backup* mode, this mode spends two more round-trip times after a handover before the new MPTCP subflow is established and data can be sent.

Figures 2.2(a) and 2.2(b) depict respectively how the *Backup* and *Single-Path* modes react to a WiFi loss. At first sight there is not a lot of differences except that the *Backup* mode requires to perform a three-way handshake on the 3G interface at connection establishment time. This causes the interface to go to the CELL_DCH state and thus requires it to consume a lot of energy. This is a major drawback. however the advantage is that it can more quickly react to WiFi failures. Indeed, depending on the handover delay, i.e., delay between the establishment of the connection and the time when the WiFi is lost, the 3G interface can still be in CELL_DCH and thus does



(a) *Backup* mode. (b) *Single-Path* mode.

Figure 2.2: The differences between the *Backup* and *Single-Path* modes.

not suffer from the 2 seconds delay before actually being able to send new data which the *Single-Path* mode always suffers from.

It is therefore not easy to decide between the two modes. An element that could be used to decide is the overall consumption of the device. Indeed, if the phone is currently used by the user, the screen is on, CPU cycles are used, etc. This already consumes energy and thus the *Backup* mode could be used in this case. Otherwise, for background traffic, i.e., when the screen is off, the *Single-Path* mode could be used. In this case the 2 seconds delay will not impact the user experience as it will not be aware of the handover.

2.3 Evaluation

In this section we provide a first evaluation of the performance trade-offs when using the three handover modes presented in the previous section. Rather than providing an actual and detailed analysis of how MPTCP behaves in various of mobile environments, we show empirically the benefit of using MPTCP in a real and specific environment. The results do not ensure the benefit of MPTCP but rather give hints on how MPTCP could improve

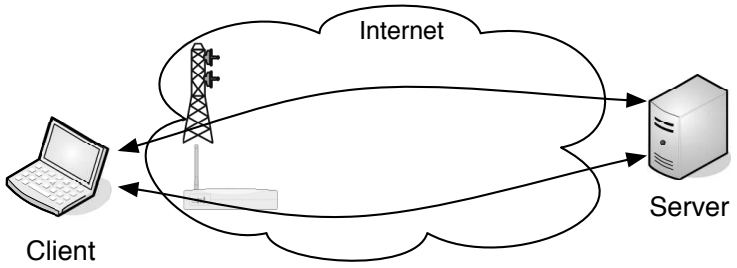


Figure 2.3: The setup used throughout the experiments.

the user experience while being mobile.

Our handover experiments start with a client connected to both WiFi and 3G networks. After five seconds we disable the WiFi interface on the ADSL router. This emulates a mobile user moving out of WiFi coverage: the client discovers the failure of the WiFi network and switches to only using its 3G connection. This vertical handover from WiFi to 3G introduces an abrupt change in path qualities.

We performed our measurements in real networks. Using such networks is more difficult than performing lab measurements, but much more realistic since it shows the ability of MPTCP to function in existing networks. Our measurement setup (see Figure 2.3) consists of a client (i3 @2.5GHz 4GB RAM) connected to both a commercial 3G network and through a WiFi interface to an ADSL broadband access provider. The 3G network offers a bandwidth between 1 and 2 Mbps and suffers from huge bufferbloat. We observed up to 2 seconds RTT when sending bursts of data. The ADSL bandwidth is around 8 Mbps downstream and 450 Kbps upstream with a minimum RTT of about 30ms. The client connects to a server (Xeon @2.67GHz 4GB RAM) located in a public-hosting server farm. Both the client and the server run our MPTCP kernel implementation (version 2.6.38). To simulate the WiFi failure we automatically shutdown the WiFi access point. A more realistic scenario would have involved actual mobile nodes. This however limits the automatization of the measures or requires equipment that we did not have.

In the rest of this section, we explore the impact of a handover on MPTCP connections using the different modes on both bulk data transfers and a VoIP application.

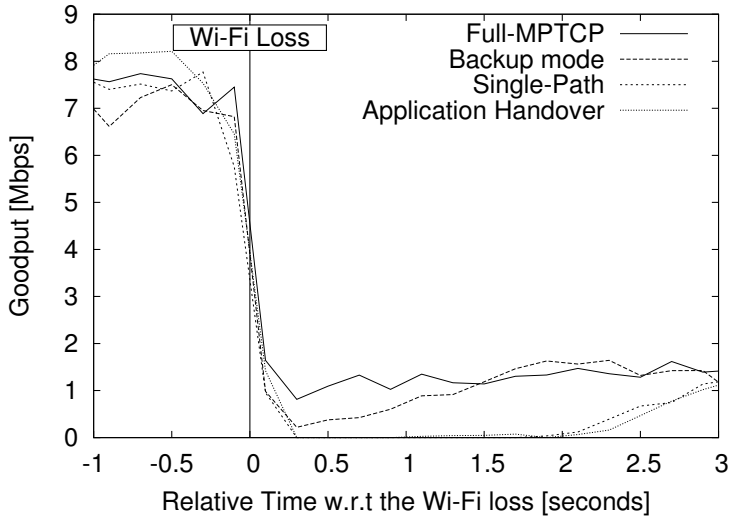


Figure 2.4: It takes up to three seconds to recover from an address loss when performing handover with MPTCP.

2.3.1 Download goodput

We evaluate the evolution of the goodput for a simple HTTP application during vertical handover. Figure 2.4 shows this goodput averaged over 200ms intervals over 20 measurements for the different handover modes. The x-axis shows the offset compared to the time when the client looses its WiFi connection (the point 0).

For comparison purposes, Figure 2.4 also shows the goodput of an *Application-level Handover* which is a modified HTTP client (running over regular TCP) that monitors the changes in the routing table to detect the failure of the WiFi interface. Upon detection of the failure, this application restarts the HTTP download by using the *HTTP Range* header. As described in Section 1.2.2, such handover requires significant changes to the application, and is specific to the HTTP/1.1 protocol [FGM+99]. In contrast, MPTCP does not require any application-level modification. Further, even if all client applications use HTTP Range, a brief study we performed shows that only 39% of the top ten thousand Alexa websites support this feature. Although feasible in theory, application-level handover cannot always be used in practice.

We can see in Figure 2.4 that when the WiFi network becomes unavailable, the *Backup* mode behaves similarly to *Full-MPTCP* which is normal. The TCP subflow on the 3G interface is already established in both cases. The difference between the two modes comes from the fact that in *Backup* mode only the three segments belonging to the three-way handshake have

been sent on the 3G interface while with the *Full-MPTCP* mode data segments have also been transmitted on this interface. As defined in Section 1.3, each subflow maintains its own congestion window. As we use an application that tries to push as much data as possible on the network, i.e., the application is network limited, data has been received/sent over both subflows for the *Full-MPTCP* mode such that both subflows have an accurate view their path. *Full-MPTCP* recovers therefore more quickly because, at the time of the failure the congestion window on the 3G subflow is larger than in the *Backup* mode, where the congestion window still has the initial value (10 MSS).

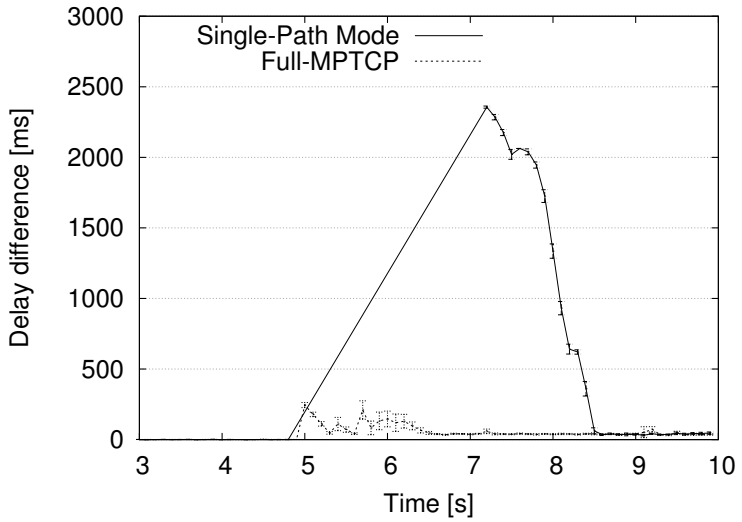
The *Single-Path* mode and *Application-level Handover* are impacted by the time the 3G interface takes to go from IDLE to the CELL_DCH state. Note that the *Backup* mode can also suffer from this delay, however in our scenario the 3G interface is already in the CELL_DCH state when the handover is performed. Without this impact the *Single-Path* mode and *Application-level Handover* behave like the *Backup* mode. The *Single-Path* mode and *Application-level Handover* both need to perform a three-way handshake and then respectively reinject and send data into this new subflow/connection, starting with an initial congestion window. After three seconds, all modes reach the average download speed on our 3G network.

We also performed measurements where we forced the 3G interface to remain in the CELL_DCH state by using a background traffic with regular ping probes. We found that there were not much differences between the *Backup* and *Single-Path* modes. In this case, the performance impact of the three-way handshake is negligible on average.

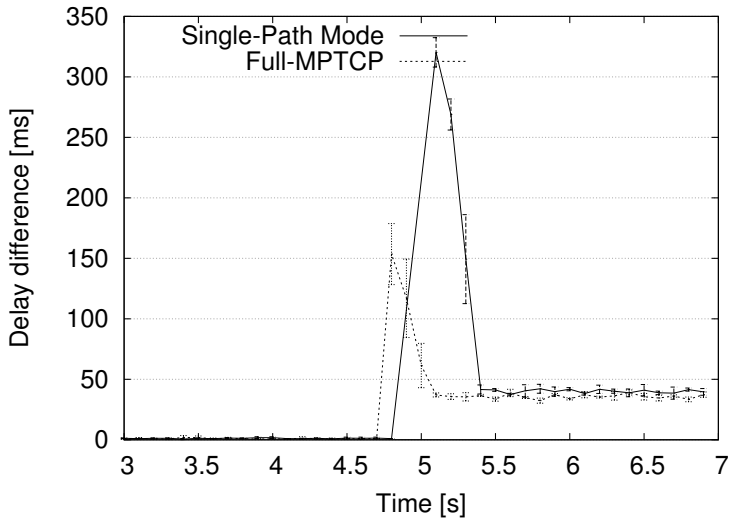
2.3.2 Application delay

Variation in application delay is an important metric for streaming or VoIP applications. We measure the application delay by sending blocks of data, tagged with a timestamp. Upon reception of each block, we store the transmission timestamp together with the timestamp at the receiver-side. The evolution of the difference between these timestamps gives us the variation in application delay.

We performed measurements during which the server was transmitting at 500Kbps to the client. The application used is therefore application limited (rate is lower than the available rate on both WiFi and 3G) rather than network limited as for the previous experiment. Initially WiFi and 3G are enabled and data segments are sent according to the specified handover mode. After around 5 seconds, the WiFi access-point is disabled and traffic has to switch over to the 3G interface.



(a) The 3G modem takes about two 2 seconds to switch from idle to operational-state.



(b) When forcing 3G to stay enabled, the impact of *Single-Path* is less important.

Figure 2.5: Evolution of the application delay.

Figures 2.5(a) and 2.5(b) show the impact on the application delay when handing the traffic over from WiFi to 3G. The measures are averaged over 100ms intervals over 20 runs of the same experiments. We can observe on the figure that the results contain an offset depending on the mode used (especially in Figure 2.5(b)). This is due to the fact that the access point shutdown and the detection on the client side is not deterministic and can take more time for some experiments than for others.

In this experiment, we are application limited and due to the MPTCP scheduler (see Section 1.3.3), the *Backup* and *Full-MPTCP* modes behaves similarly. Indeed, in the case of *Full-MPTCP* almost no data will be sent on the 3G subflow. A sending rate of 500Kbps does not fill the pipe over the WiFi interface. Since MPTCP prefers the WiFi path as it has the lowest RTT, no traffic is sent over the 3G network. The *Backup* mode is therefore not shown since it is similar to *Full-MPTCP*.

As in *Single-Path* mode no subflow is established over 3G prior to the vertical handover event. The 3G interface is idle and thus first needs to becomes active before establishing the new subflow. This takes up to 2 seconds and the impact on the application delay can be observed in Figure 2.5(a). When the 3G interface is forced to remain active, the impact of establishing the new subflow over *Single-Path* is less important as can be seen in Figure 2.5(b). *Full-MPTCP* has a slight peak during handover due to two reasons: First, the client needs to announce the lost WiFi-interface by using the REMOVE_ADDR option. Second, because only a small number of data segments were sent through the 3G interface, its congestion window is not yet large enough to sustain a sending rate of 500 Kbps. The application delay is higher during the slow-start phase until the congestion window allows a steady rate of 500Kbps.

2.3.3 Impact on existing applications

Since MPTCP does not change the socket API, it can be used transparently by any TCP application. Skype is a commercial Voice over IP application that is able to operate over both TCP and UDP. We experiment with Skype – that has very tight constraints on packet-level delays – to showcase a worst-case for the MPTCP handover. We do not imply that Skype should be run over TCP or MPTCP – it is much better to run Skype over unreliable transports such as RTP or UDP if possible.

For our experiment, we force Skype to pass through an MPTCP-enabled HTTP-proxy by blocking UDP and other TCP ports on the client's firewall. Otherwise Skype will by default try to use UDP or do regular TCP over the public Skype-servers. We use the Skype API to play a recorded file and record the received signal. Around the 7^{th} second we turn down the WiFi access point and MPTCP seamlessly performs the handover to the 3G interface without any impact on the application.

We compare the impact on the voice signal during handover from WiFi to 3G with the different handover modes. Figure 2.6 shows the amplitudes of the original signal and the signals impacted by the handover. When using MPTCP in either handover modes, the Skype-session does not break and

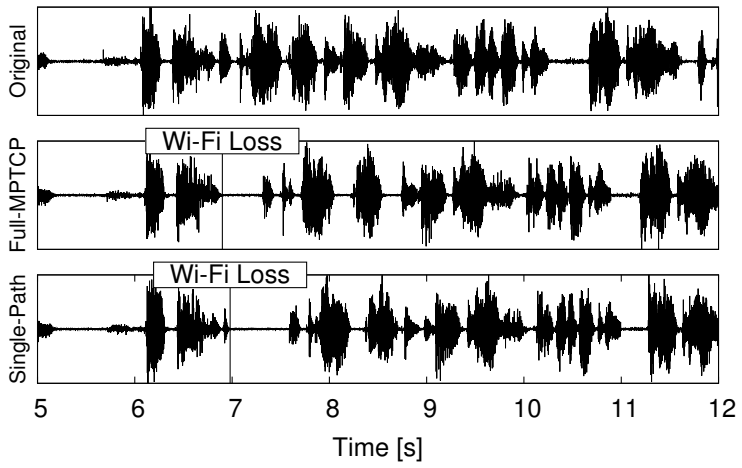


Figure 2.6: During the failover to 3G a short moment of silence happens during the Skype-call.

the call continues. A short moment of silence follows the loss of the WiFi interface. This is because MPTCP needs to detect the failure and then reinject the lost data segments in another subflow (*Full-MPTCP*) or in a newly established subflow (*Single-Path*). Hence, Skype plays the received voice signal with some additional delay but resynchronizes the call after some time.

2.3.4 Summary of experiments

We have seen that MPTCP allows unmodified applications such as Skype to continue during a handover from WiFi to 3G. To our best knowledge, this is the first time such handover has been tested on real networks and with real applications.

Using the 3G interface is costly in terms of battery-lifetime, so using *Full-MPTCP* mode is not the best option for users willing to save battery life. Of course, other mobile devices might have different energy consumptions, and the results might be different. On the other hand, *Full-MPTCP* mode offers the smoothest handover. Using the *Backup* mode brings modest performance improvements over *Single-Path* mode during handover when the connection is long enough to allow the 3G interface to go to sleep. In such cases the *Single-Path* mode uses less energy and is thus preferable. If connections are shorter than 10 seconds and a handover happens, the *Backup* mode is preferable as it gives good performance at modest energy cost.

Given this initial exploration, an interesting question is, how mobile operating systems can optimize user experience and battery-lifetime simultaneously. One option is to always use *Full-MPTCP* mode when the user is actively using the phone (i.e., the mobile is in active mode). When the mobile is sleeping, it makes sense to use less energy hungry modes such as *Single-Path* mode.

2.4 Improving Multipath TCP

During our measurements we stumbled sometimes upon unexpected results. In the *Backup* mode the second subflow ended up unused after the first one was lost, and large application delays were observed in *Single-Path* mode. Both issues originated from the loss of the REMOVE_ADDR option which is sent unreliably. Indeed, the REMOVE_ADDR and ADD_ADDRESS options are typically sent on separate TCP ACKs. MPTCP does not treat duplicate ACKs containing such options as a signal of congestion [FRHB13] and thus these options may get lost in the network. In this section we discuss these results and show how we can address them by changing MPTCP to send these options reliably.

Observations Losing the REMOVE_ADDR option has an impact on MPTCP's performance. When receiving the REMOVE_ADDR option, the peer closes the affected subflow and reinjects all unacknowledged data on the second subflow. However, if the REMOVE_ADDR option is lost, the peer continues to send data on all existing subflows, i.e., in this case the first and second subflow. Reinjection of data from the affected subflow will only occur after a *Retransmission Timeout* (RTO). On congested wireless networks the RTO can easily shoot up to a few seconds, which significantly affects the handover process. Furthermore, the initial subflow will remain active until the expiration of the maximum retransmission-timeout (between 13 and 30 minutes), using precious server resources.

We observed the impact on application delay when losing the REMOVE_ADDR option. We found as expected that in the *Backup* mode, if the REMOVE_ADDR option is lost, the server is not aware that it has to start using the backup-subflow and thus no data is ever received after losing the WiFi interface. Figure 2.7 shows the impact on the performances when the REMOVE_ADDR is lost when the *Backup* mode is used. The figure shows similarly to Figure 2.4 the application goodput averaged over 500ms periods over 10 measurements. We can see that when the REMOVE_ADDR is lost the goodput drops to 0. Technically the connection could only recover after the maximum retransmission-timeout when the WiFi subflow is destroyed

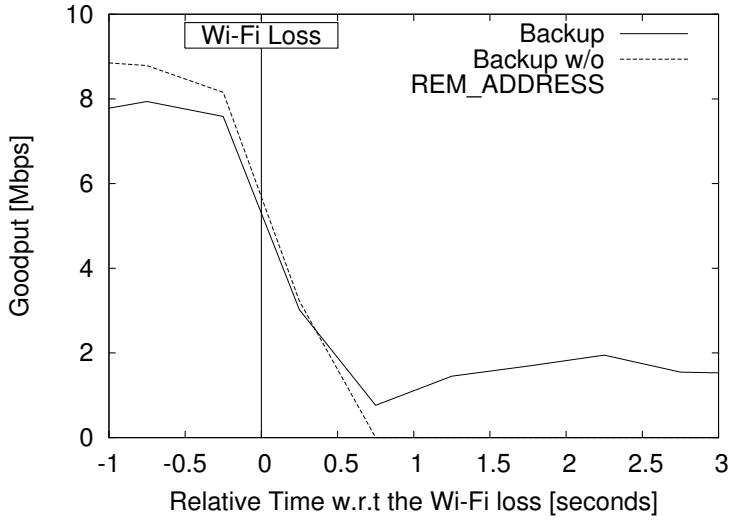


Figure 2.7: The loss of a REMOVE_ADDR during a handover significantly impacts the performances.

on the server. In this case, the latter is able to further send data on 3G since it is the remaining subflow. If the REMOVE_ADDR option is lost in *Full-MPTCP* mode, the server has to wait until the expiration of the RTO on the subflow passing by the WiFi-interface. And only after this RTO the data will be reinjected over the second subflow. This increases the handover delay.

The current MPTCP specification does not include any mechanism to ensure a reliable delivery of the REMOVE_ADDR and ADD_ADDRESS options. In our current implementation, we use one spare bit in these options that serves as an *echo* bit. We use it as follows. When a host loses an address it sends the REMOVE_ADDR option in a duplicate ACK as described in Section 1.3.4 and starts its retransmission timer. The REMOVE_ADDR option is also automatically added to each outgoing segment. When a host receives a segment containing the REMOVE_ADDR option it then immediately replies with a duplicate ACK that contains the same option with the *echo* bit set. Upon reception of this ACK, the initiator of the REMOVE_ADDR is now certain that it was received. We implemented this design in our implementation which consisted of around hundred lines of code; all the experiments we have run in Section 2.3 use this optimization.

2.5 Related work

Over the last decade, several research efforts have been conducted to enable end hosts to support mobility for applications in a transparent manner. At the different layers of the OSI model, protocols such as the *Stream Control Protocol* (SCTP) [SXM+00], Mobile IP [Per10] and MIPv6 [PJA11] have been trying to allow a mobile node to seamlessly perform vertical handover.

Snoeren et al. proposed Migrate TCP [SB00] that uses TCP options and DNS updates to migrate the endpoint of a TCP session to a different address. This could be readily used for 3G/WiFi offloading. Raiciu et al. have argued in [RNBH11] that MPTCP is a better solution because it can use all the interfaces at once which provides increased performance and throughput.

Offloading 3G to WiFi is not a novel idea. Wiffler [BMV10] presents a study of 3G and WiFi coverage while driving and shows how offload can be implemented by changing the applications. Our solution is more general since it enables unmodified applications to benefit from offloading.

2.6 Conclusion

In this chapter, we have carried out an experimental investigation of Multipath TCP in the presence of WiFi and 3G. We have proposed and evaluated three handover modes: *Full-MPTCP*, *Backup* and *Single-Path*. Our experiments in commercial wireless networks demonstrate that MPTCP can play a role for mobile users and also WiFi/3G convergence today. Our measurements show that MPTCP can quickly recover from a WiFi loss in presence of a 3G interface with only a small impact on the application delay and goodput. Our experiments with Skype demonstrate that existing unmodified applications already benefit from MPTCP.

Chapter 3

Detecting middleboxes interference

3.1 Introduction

In Chapter 1, we have seen that the architecture of the Internet does not follow the end-to-end principle anymore. The network is composed of middleboxes that interfere with the end-hosts' TCP connections. Although these middleboxes are supposed to be transparent to the end user, experience shows that they have a negative impact on the evolvability of the TCP/IP protocol suite [HNR+11]. For example, after more that ten years of existence, SCTP [SXM+00] is still not widely deployed, partially because many firewalls and NAT may consider SCTP as an unknown protocol and block the corresponding packets. We have also seen in Chapter 1 that middleboxes have heavily influenced the design of MPTCP.

Recent papers have shed the light on the deployment of those middleboxes. For instance, Sherry et al. [SHS+12] obtained configurations from 57 entreprise networks and revealed that they can contain as many middleboxes as routers. Wang et al. [WQX+11] surveyed 107 cellular networks and found that 82 of them used NATs. D'Acunto et al. [DCVS13] analyzed *Peer-to-Peer* (P2P) applications and found that 88% of the participants in the studied P2P network were behind NATs.

Despite of their growing importance in handling operational traffic, middleboxes are notoriously difficult and complex to manage [SHS+12]. One of the causes of this complexity is the lack of debugging tools that enable operators to understand where and how middleboxes interfere with packets. Many operators rely on `ping`, `traceroute`, and various types of `show` commands to monitor their networks. These tools use packets that can be dealt

with differently from actual application packets [PCVB13].

In this chapter, we propose, validate, and evaluate `tracebox`. `tracebox` is a `traceroute` [V. 89] successor that enables network operators to detect which middleboxes modify packets on almost any path. `tracebox` allows one to identify various types of packet modifications and can be used to pinpoint where a given modification takes place. We present several use cases based on a deployment of `tracebox` on the PlanetLab testbed.

The remainder of this chapter is organized as follows: Section 3.2 describes the operations of `tracebox` and how it is able to identify middleboxes along a path. Section 3.3 describes the implementation of `tracebox` as well as the API it provides to the user. Section 3.4 analyses three use cases from the deployment of `tracebox` on PlanetLab. Section 3.5 shows how `tracebox` can be used to debug networking problems. Section 3.6 compares `tracebox` regarding the state of the art. Finally, Section 3.7 concludes the chapter and discusses potential further work.

3.2 Tracebox

To detect middleboxes, `tracebox` uses the same incremental approach as `traceroute`, i.e., sending probes with increasing TTL values and waiting for ICMP `time-exceeded` replies. While `traceroute` uses this information to detect intermediate routers, `tracebox` uses it to infer the modification applied on a probe by an intermediate middlebox.

The `TCPExposure` software developed by Honda et al. [HNR+11] is probably the closest to `tracebox`. The study realized with this software revealed various types of packet modifications with specially crafted packets to test for middlebox interference. However, this tool is limited to specific paths as both ends of the path must be under control. This is a limitation since some middleboxes are configured to only process the packets sent to specific destination or ports. On the contrary, `tracebox` does not require any cooperation with the servers. It allows one to detect middleboxes on any path, i.e., between a source and any destination.

`tracebox` therefore brings two important features:

Middlebox detection

`tracebox` allows one to easily and precisely control all probes sent (IP header, TCP or UDP header, TCP options, payload, etc.). Further, `tracebox` keeps track of each transmitted packet. This permits to compare the quoted packet sent back in an ICMP `time-exceeded` by an intermediate router with the original one. By correlating the different modifications, `tracebox` can infer the presence of middleboxes.

Middlebox location
Using an iterative technique (in the fashion of `traceroute`), `tracebox` can approximately locate, on the path, where packet modifications occurred and so the approximate middleboxes position.

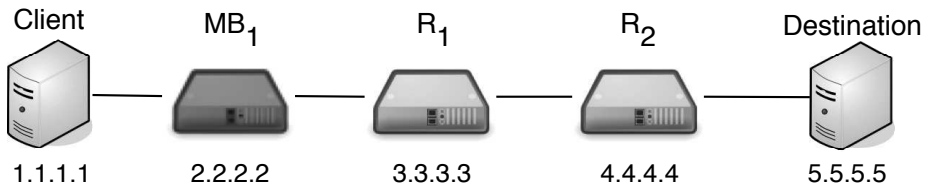
3.2.1 Packets Modification Detection

When an IPv4 router receives an IPv4 packet whose TTL is going to expire, it returns an ICMPv4 `time-exceeded` message that contains the modified packet. According to RFC792, the returned ICMP packet should quote the IP header of the original packet and the first 64 bits of the payload of this packet [Pos81]. When the packet contains a TCP segment, these first 64 bits correspond to the source and destination ports and the sequence number. In 1995, the situation changed as routers were required to quote the entire IP packet encapsulated in the ICMP message [Bak95].[1] RFC1812 [Bak95] recommended to quote the entire IP packet in the returned ICMP, but this recommendation has only been recently implemented on several major vendors' routers at the same time as the deployment of the MPLS extensions for ICMP [BGTP07]. Discussions with network operators showed that recent routers from Cisco (running IOX), Alcatel Lucent, HP, Linux, and Palo-Alto firewalls return the full IP packet. In the remainder of this chapter, we use the term *Full ICMP* to indicate an ICMP message quoting the entire IP packet. We also use the term RFC1812-compliant router to indicate a router that returns a *Full ICMP*.

`tracebox` exploits these returned IP packet to detect some modifications performed on a path. By analyzing the returned quoted packets, `tracebox` is able to detect various modifications performed by middleboxes and routers. This includes changes in the Differentiated Service field and/or the Explicit Congestion Notification bits in the IP header, changes in the IP identification field, packet fragmentation, and changes in the TCP sequence numbers. Further, when `tracebox` receives a *Full ICMP*, it is able to detect more modifications such as changes to the TCP acknowledgement number, the TCP window, the removal/addition of TCP options, payload modifications, etc.

`tracebox` also allows for more complex probing techniques requiring to establish a connection and so multiple probes to be sent, e.g., to detect segment coalescing/splitting, Application-level Gateways, etc. In this case `tracebox` works in two phases: the *detection* and the *probing* phases. During the detection phase, `tracebox` sends probes by iteratively increasing the TTL until it reaches the destination. This phase allows `tracebox` to

[1] It is worth to notice that ICMPv6 has the same recommendation [CDG06].



(a) topology

```
# tracebox -p 'IP / TCP / mss(9000)' -n 5.5.5.5
tracebox to 5.5.5.5 (5.5.5.5): 30 hops max
1: 3.3.3.3 TCP::SequenceNumber
2: 4.4.4.4 IP::TTL IP::CheckSum TCP::CheckSum TCP::SequenceNumber
           TCPOptionMaxSegSize::MaxSegSize
3: 5.5.5.5
```

(b) output

Figure 3.1: `tracebox` example.

identify RFC1812-compliant routers. During the probing phase, `tracebox` sends additional probes with TTL values corresponding to the previously discovered RFC1812-compliant routers. This strategy allows `tracebox` to reduce its overhead by limiting the number of probes sent.

3.2.2 Example

Figure 3.1(a) shows a simple network, where MB_1 is a middlebox that changes the TCP sequence number and the MSS size in the TCP MSS option but does not decrement in this case the TTL. R_1 is an old router while R_2 is a RFC1812-compliant router. The server always replies with a TCP reset. The output of running `tracebox` between "Source" and "Destination" is shown in Figure 3.1(b). It shows that `tracebox` is able to effectively detect the middlebox interference but not its exact location. Indeed, as R_1 does not reply with a *Full ICMP*, `tracebox` can only detect the TCP sequence number change when analyzing the reply sent by R_1. Nevertheless, when receiving the *Full ICMP* message from R_2 which contains the complete IP and TCP headers, `tracebox` is able to detect that a TCP option has been changed upstream of R_2. At the second hop, `tracebox` shows additional modifications on top of the expected ones. The TTL and IP checksum are modified by each router and the TCP checksum modification results from the modifications of the header.

3.2.3 Limitations

The detection of middleboxes depends on the reception of ICMP messages. `tracebox` suffers therefore from some limitations linked to ICMP:

Filtering
Networks can filter ICMP messages therefore preventing `tracebox` to be used. If the router downstream of a middlebox does not reply with an ICMP message, `tracebox` will only be able to detect the change at a downstream hop similarly as in the example of Figure 3.1.

Rate limit
ICMP generation is often done in software which can be costly for routers. Routers therefore sometimes limit the rate at which they generate ICMP messages. `tracebox` solves this issue by generating several probes for each TTL value if no reply has been received.

Servers
Servers do not always reply with an ICMP message[2] (as in Figure 3.1). In this case it is impossible to detect middleboxes upstream of them.

Revert modification
Some middleboxes, e.g., NATs, revert the modification they apply on the outgoing packets in the incoming ICMP messages. This implies that, when inspecting the received ICMP message, `tracebox` would not be able to detect the modification. We explore this issue in more detail in Section 3.5.3 and propose a solution to detect NATs.

3.3 Implementation

`tracebox` [Det13] is implemented in C++ in about 2,000 lines of code and embeds LUA [IdFF96] bindings to allow a flexible description of the probes as well to ease the development of more complex middlebox detection scripts. LUA is a lightweight multi-paradigm programming language designed as a scripting language. LUA is widely used by game programmers as it is easily embedded in other programs.

`tracebox` aims at providing the user with a simple and flexible way of defining probes without writing many lines of code. `tracebox` indeed allows to use a single line to define a probe like Scapy [Bio]. `tracebox` uses the libcrafter library [Pel13] as backend to represent probes. Each probe is defined by using the following LUA syntax:

[2] Linux does not by default generate an ICMP for incoming packets with TTL=1.

```
<Packet> ::= <Protocol> <Concat> <Protocol>
<Packet> ::= <Packet> <Concat> <Protocol>
<Concat> ::= '/' | '+' | '.'
```

`Protocol` represents an OSI layer such as IP or TCP but also extensions to these layers such as TCP options. The combination of two `Protocol`s creates a `Protocol` which can be further combined with additional `Protocol`s. A probe is a `Protocol`. Several operations can combine `Protocol`s and `Packet`s: `/`, `+` and `.`. In this chapter the notation `/` is used.

`tracebox` provides a complete API to easily define IPv4/IPv6 as well as TCP, UDP, ICMP headers and options on top of a raw payload. The API is based on functions that return a `Protocol` object. The following functions are available[3]:

- `ip{}` corresponds to IPv4. Optional arguments are:
  - `dst` specifies the destination as either a DNS name or an IP address in the string format. If not set the destination used as argument of `tracebox` is used.
  - `id` specifies the IP identification. By default it is set to `0` on Linux and to a random value on Mac OS X. The latter's IP stack generates a random value if not set. Generating it in `tracebox` allows to avoid false positives, i.e., detecting a middlebox that randomizes the IP ID when there is actually none.
  - `ecn` and `dscp` are used to specify the Explicit Congestion Notification [RFB01] and Differentiated Services fields of the IP header.
  - `proto` specifies the protocol number. If not set, the protocol number of the upper `Protocol` is automatically used.
- `ipv6{}` corresponds to IPv6. Optional arguments are:
  - `dst` behaves similarly to `dst` of `ip{}`.
  - `tc` and `flowlabel` specify the Traffic Class and the Flow Label of the IPv6 header.
- `tcp{}` corresponds to TCP. Optional arguments are:
  - `src` and `dst` respectively specify the source and destination ports. By default a random value is used for the former and `80` is used for the latter.

[3]LUA uses the notation `<name>{<args>}` to describe a variable length argument procedure while `<name>(<args>)` for mandatory arguments. This is in fact a syntactic sugar where the notation `<name>{<args>}` corresponds to `<name>({<args>})`, i.e., a procedure with a dictionary as mandatory argument.

  - `seq` and `ack` respectively specify the sequence and acknowledgment numbers. By default a random value is used for the former and `0` for the latter.
  - `win` specifies the window size. By default `5840` is used.
  - `flags` specifies the TCP flags using an integer representation. By default `0x02` is used which corresponds to the SYN flag.
- `raw('<string>')` corresponds to the payload. There are no optional arguments.

Note that the IPv4/IPv6 source address is automatically set using the address of the default interface or of the specified interface.

On top of these functions, the API provides macros that allow to simplify the description of probes. For example, the macro `TS`, which corresponds to `NOP / NOP / timestamp{}`, is the combination of two NOP options (`NOP / NOP`) followed by the TCP Timestamp option. This macro ensures that the TCP option is aligned on 4 bytes (the TCP Timestamp option is 10 bytes long).

We list in the following three examples of probes to show the potential probe spaces of `tracebox`.

- `IP / TCP / WSCALE / MPCAPABLE` corresponds to a SYN probe to port 80 over IPv4. The probe also contains the Window Scale TCP option as well as the MP_CAPABLE option used by MPTCP. The probe `ip{} / tcp{} / wscale{} / NOP / mpcapable{}` has the same effect.
- `IPv6 / udp{dst=56987} / raw('payload test')` corresponds to a UDP probe with destination port 56987 over IPv6 containing the string `'payload test'` as payload.
- `IP / tcp{src=2345,flags=0x12,ack=345678}` corresponds to a SYN+ACK probe where the source port is set to 2345 and the acknowledgement number is set to 345678.

`tracebox` provides an API for LUA scripting. The API allows one to write scripts in order to generate complex middlebox detection techniques. As of this writing, the API does not include an IP/TCP stack. However, the idea behind the scripting feature is to allow scripts to act as a regular application. Currently, the scripts handle the generation of SYN and call `tracebox()` which takes a `Packet` as argument and optionally a callback function. The latter is called for each ICMP messages received and has the detected modification as parameter. `tracebox()` also returns the reply from the server. The scripts therefore receives the SYN+ACK and have to further continue emulating a simple IP/TCP stack. An example of a such script is

```
# tracebox -v -n -p 'IP/TCP/mss(1460)/MPCAPABLE/WSCALE' bahn.de
tracebox to 81.200.198.6 (bahn.de): 64 hops max
1: 130.104.228.126
2: 130.104.254.229 IP::TTL IP::CheckSum
3: 193.191.3.85 IP::TTL IP::CheckSum
4: 193.191.16.21 IP::TTL IP::CheckSum
5: 195.66.224.124 IP::TTL IP::CheckSum
6: 145.254.5.226 IP::TTL IP::CheckSum
7: 88.79.13.62 IP::TTL IP::CheckSum
8: 81.200.194.234 IP::TTL IP::CheckSum
9: 81.200.197.9 IP::TTL IP::CheckSum
10: 81.200.198.6 TCP::CheckSum IP::TTL IP::CheckSum
  TCPOptionMaxSegSize::MaxSegSize -TCPOptionMPTCPCapable
  -TCPOptionWindowScale
```

Figure 3.2: `tracebox` output on today's Internet.

provided in Section 3.5. The scripts are part of the `tracebox` distribution. They are open-source and publicly available [Det13].

To verify the ability of `tracebox` to detect various types of middlebox interference, we developed several Click elements [KMC+00] modeling middleboxes. We wrote Click elements that modify various fields of the IP or TCP header, elements that add/remove/modify TCP options and elements that coalesce or split TCP segments. These elements have been included in a python library [Hes13] that allows to easily describe a set of middleboxes and generates the corresponding Click configuration. This library is used as unit tests to validate each new version of `tracebox`.

Figure 3.2 shows the output of `tracebox` on today's Internet. The source is located in a University network and the destination is the website of the public transportation service of Germany. The probe sent is a TCP SYN probe that contains the MSS, MPTCP and Window Scale TCP options. We can see that at each router hop the TTL and the IP checksum are modified, this is a normal router behavior and must be ignored. The interesting element is located at the last hop. We can see that the server (or load balancer) replied with a *Full ICMP*. This indicates that there is a middlebox in front of the server that modifies the value of the MSS option and removes the MPTCP and Window Scale options. Running `tracebox` with the `-v` (verbose) option revealed that the MSS value was reduced from 1460 to 1380 bytes.

Figure 3.3: The geolocation of the VPs used during the experiments.

3.4 Validation & use cases

In this section, we validate and demonstrate the usefulness of `tracebox` based on three use cases. We first explain how we deploy `tracebox` on the PlanetLab testbed (Section 3.4.1), next we assess the coverage of `tracebox` (Section 3.4.2) and finally discuss our use cases (Sections 3.4.3, 3.4.4, and 3.4.5).

3.4.1 PlanetLab deployment

We deployed `tracebox` on PlanetLab, using 72 machines as *Vantage Points* (VPs). Figure 3.3 shows the approximate geolocation of each VP. Each VP had a target list of 5,000 DNS names build with the top 5,000 Alexa web sites, i.e., the top visited websites on the Internet. Each VP used a shuffled version of the target list. DNS resolution was not done before running `tracebox`. This means that, if each VP uses the same list of destination names, each VP potentially contacted a different IP address for a given web site due to the presence of load balancing or Content Distribution Networks. Our dataset was collected during one week starting on April 17$^{\text{th}}$, 2013.

In this chapter, we focus on analyzing interferences between middleboxes and (MP)TCP. In theory, PlanetLab is not the best place to study middleboxes because PlanetLab nodes are mainly installed in research labs with unrestricted Internet access. Surprisingly, we noticed that seven VPs, from the 72 considered for the use cases, automatically removed or changed TCP options at the very first hop. They replaced the Multipath TCP,

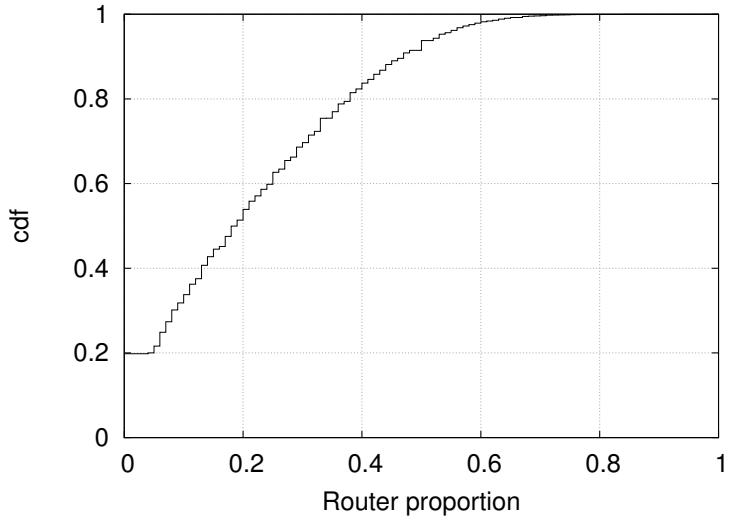


Figure 3.4: Proportion of RFC1812-compliant routers on a path.

MD5 [Hef98], and Window Scale [JBB92] options with NOP options and changed the value of the MSS option. We also found that two VPs always change the TCP Sequence number.

3.4.2 RFC1812-compliant routers

`tracebox` keeps track of each original packet sent and makes a comparison with the quoted IP packet when the ICMP `time-exceeded` message is received. Further, `tracebox` can potentially detect more middleboxes when routers return a *Full ICMP* packet. `tracebox`'s utility clearly increases with the number of RFC1812-compliant routers. Figure 3.4 and Figure 3.5 provide an insight of the proportion of RFC1812-compliant routers and their locations.

In particular, Figure 3.4 gives the proportion of RFC1812-compliant routers (the horizontal axis) as a CDF. A value of 0, on the horizontal axis, corresponds to paths that contained no RFC1812-compliant router. On the other hand, a value of 1 corresponds to paths composed of only RFC1812-compliant routers. Looking at Figure 3.4, we observe that, in 80% of the cases, a path contains at least one router that replies with a *Full ICMP*. In other words, `tracebox` has the potential to reveal more middleboxes in 80% of the cases.

Figure 3.5 estimates the position of the RFC1812-compliant routers in the probed paths. It provides the distance between the VP and the RFC1812-compliant routers on a given path. Note that, on Figure 3.5, the

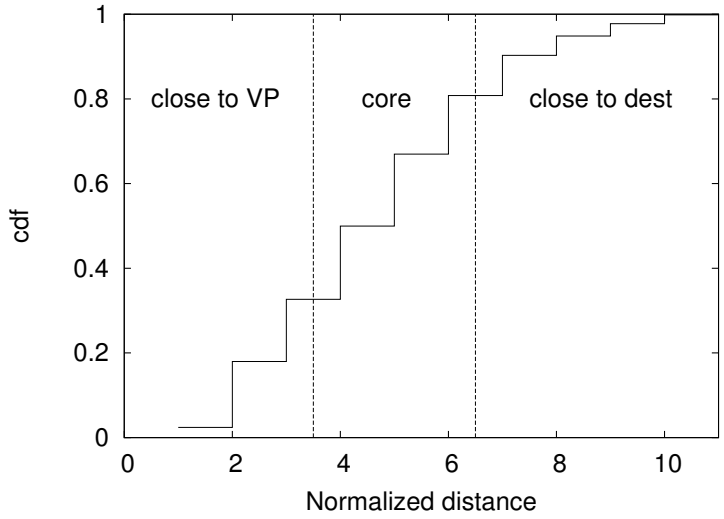


Figure 3.5: Normalized distance from VP to RFC1812-compliant router.

X-Axis (i.e., the distance from the VPs) has been normalized between 1 and 10. Distances between 1 and 3 refer to routers close to the VP, 4 and 6 refer to the Internet core while, finally, distances between 7 and 10 refer to routers closer to the `tracebox` targets. The widespread deployment of RFC1812-compliant routers in the Internet core is of the highest importance since `tracebox` will be able to use these routers as "mirrors" to observe the middlebox interferences occurring in the access network [WQX+11].

Figure 3.5 shows that for 22% of the paths, the RFC1812-compliant routers are close to the VP. This is approximatively the same proportion for routers close to `tracebox` targets. However, the majority of routers (roughly 50%) returning a *Full ICMP* are located in the network core. Indeed, MPLS is likely to be used in the core network and the MPLS extension to `traceroute` requires routers to reply with a *Full ICMP*.

3.4.3 TCP sequence number interference

The TCP sequence number is not supposed to be modified by intermediate routers. Still, previous measurements [HNR+11] showed that some middleboxes change sequence and acknowledgement numbers in the processed TCP segments. As the sequence number is within the first 64 bits of the TCP header, `tracebox` can detect its interference even though there are no RFC1812-compliant routers.

We analyze the output of `tracebox` from the 72 VPs. Our measurements reveal that two VPs always modify the TCP sequence numbers. The

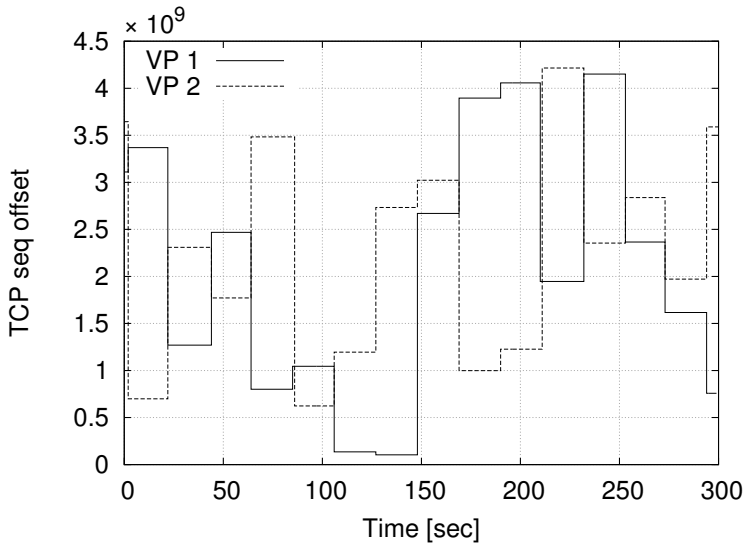


Figure 3.6: Time evolution of the TCP sequence number offset introduced by middleboxes.

position of the responsible middlebox is close to the VP, respectively the first and third hop. We suspect that the middlebox randomizes the TCP sequence number to fix a bug in old TCP/IP stacks where the *Initial Sequence Number* (ISN) was predictable [Mic99]. To prevent attackers from hijacking and injecting data – affecting host flows – firewall manufacturers preferred to solve the bug as a service, inside the firewall, provided by the network instead of waiting for a patch to the TCP/IP stack of the incriminated OS.

When used on a path that includes such a middlebox, `tracebox` can provide additional information about the sequence number randomization. Depending on the type of middlebox and the state it maintains, the randomization function can differ. To analyze it, we performed two experiments. First, we generated SYN probes with the same destination (IP address and port), the same sequence number, and different source ports. `tracebox` revealed that the middlebox modified all TCP sequence numbers as expected. A closer look at the modified sequence numbers revealed that the difference between the ISN of the probe and the randomized sequence number can be as small as 14510 and as large as 4294858380 (which corresponds to a negative difference of 108916 when the 32 bits sequence number space wrap). Our measurements show that these differences appear to be uniformly distributed for different source ports.

For our second experiment, we used `tracebox` to verify how the randomization evolves over time. For this, we sent SYN probes using the same 5-tuple and different ISN during five minutes and evaluated the evolution

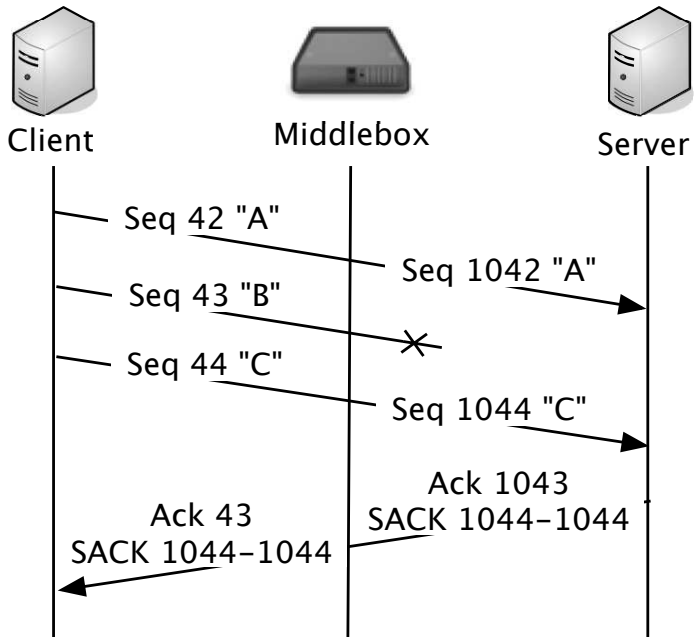


Figure 3.7: Example of invalid SACK blocks generated due to a middlebox.

of the TCP sequence number modifications. Figure 3.6 shows the offset between the sent ISN and the randomized one for two different 5-tuples. `tracebox` reveals that the two middleboxes seem to change their randomization approximatively every 20 seconds. This suggests stateful middleboxes.

As explained by Honda et al. [HNR+11], changing the TCP sequence numbers has an impact on the evolvability of the TCP protocol. Unfortunately, it has also an impact on the utilization of widely deployed TCP extensions. Consider the TCP *Selective Acknowledgement* (SACK) option [MMFR96]. This TCP option improves the ability of TCP to recover from losses. One would expect that a middlebox changing the TCP sequence number would also update the sequence numbers reported inside TCP options. This is unfortunately not true, almost 18 years after the publication of the RFC [MMFR96]. We used `tracebox` to open a TCP connection with the SACK extension and immediately send SACK blocks. `tracebox` reveals that the middlebox changes the sequence number but does not modify the sequence numbers contained in the SACK block.

Figure 3.7 shows the behavior of such a middlebox on the TCP sequence number and SACK blocks. The client sends three segments containing a sin-

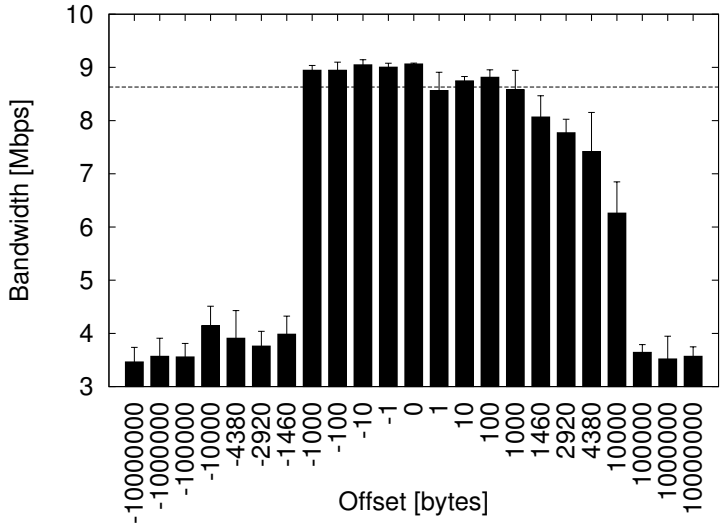


Figure 3.8: Impact on Linux performance in the presence of a middlebox that changes the sequence number.

gle byte of payload. The second byte is lost between the middlebox and the server causing the server to add a SACK block into the acknowledgement it sends back to the client in order to announce that it has already received the third byte but it is still waiting for the second one. In this scenario, the middlebox increases the TCP sequence number by 1,000 bytes causing the client to receive a SACK block that corresponds to a sequence number that it has not yet transmitted. This SACK block is invalid, but the acknowledgement is valid and correct. For the receiver, it may not be easy to always detect that the SACK information is invalid. The receiver may detect that the SACK blocks are out of the window, but the initial change may be small enough to carry SACK blocks that are inside the window.

Figure 3.8 and Figure 3.9 show the impact of such a middlebox on the performance of, respectively, the Linux and the Mac OS X TCP stacks. The figures show the application goodput averaged over 10 experiments. We model the middlebox behavior using our Click elements (that we developed to validate `tracebox`, see Section 3.3) and used the offset added to the TCP sequence number as parameter. We then plugged in a Linux and Mac OS X host and performed a performance evaluation using iperf [ipe13] by varying the offset. We limit the bandwidth to 10 Mbps and added 1% loss between the client and the server to cause the server to generate SACK blocks.

We can see, on Figures 3.8 and 3.9, that when SACK is enabled and the offset is null, the performance is better than when SACK is disabled (the dashed line). However when the offset decreases, the performance is worse than if SACK was disabled. On Linux, this is due to the fact that

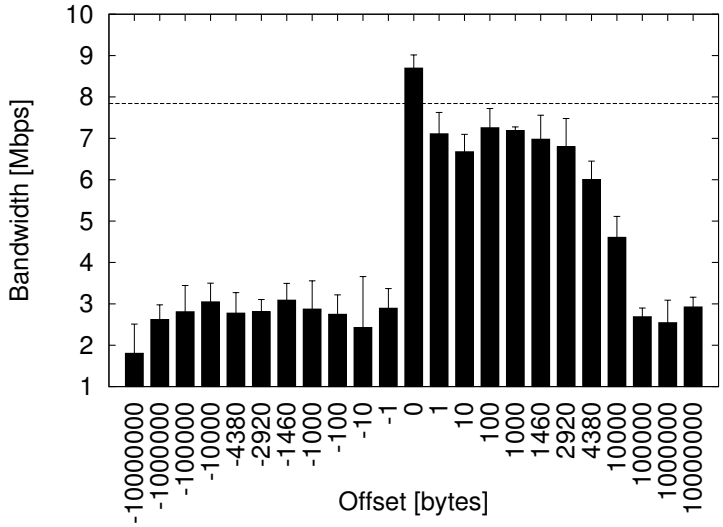


Figure 3.9: Impact on Mac OS X performance in the presence of a middlebox that changes the sequence number.

duplicate acknowledgements containing an invalid SACK block are not considered. The client waits for a complete RTO to retransmit the segments instead of doing a fast retransmit (as when SACK is disabled). This therefore dramatically decreases the performance (by more than 50%). When the offset increases, there is a higher chance that the blocks are in-window. However, once the offset is large enough, the performances are as bad as for a negative offset. On Mac OS X, the impact on performance is even worse than on Linux. As we do not have access to the implementation of SACK in Mac OS X's kernel we cannot analyze the root cause. We solved the issue on Linux by modifying the stack to consider duplicate acknowledgements when receiving out-of-window SACK blocks. This however shows the impact on performance of the end host's TCP stack when it does not handle unexpected modifications to the protocol.

3.4.4 TCP MSS option interference

Our second use case for `tracebox` concerns middleboxes that modify the TCP MSS option. This TCP option is used in the SYN and SYN+ACK segments to specify the largest TCP segment that a host sending the option can process. In an Internet that respects the end-to-end principle, this option should never be modified. In the current Internet, this is unfortunately not the case. The main motivation for changing the TCP MSS option on middleboxes is probably to fix some issues caused by other middleboxes with Path MTU Discovery [MD90]. On top of changing the MSS option,

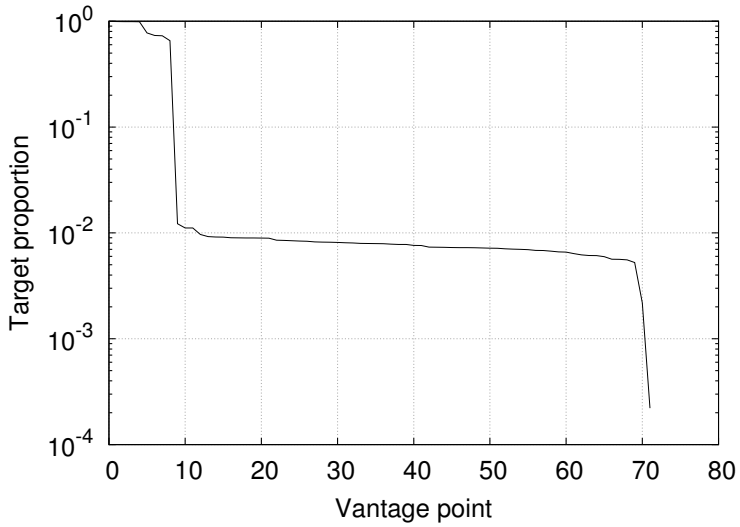


Figure 3.10: VPs proportion modifying MSS.

we also discovered middleboxes, in several ISPs, that add the option if it is missing.

Path MTU Discovery is a technique that allows a host to dynamically discover the largest segment it can send without causing IP fragmentation on each TCP connection. For that, each host sends large segments inside packets with the `Don't Fragment` bit set. If a router needs to fragment the packet, it returns an ICMP `destination-unreachable` (with code "Packet fragmentation is required but the 'don't fragment' flag is on") back to the source. The source updates its segment size. Unfortunately, some routers do not return such ICMP messages [MAF04] or some middleboxes (e.g., NAT boxes and firewalls) do not forward the received ICMP message back to the source. MSS clamping [Sav06] mitigates this problem by configuring middleboxes to decrease the size reported in the MSS option to a smaller MSS that should not cause fragmentation. With that technique or rather this *hack*, the packets' MSS is reduced by the router so that the TCP connection automatically restricts itself to the maximum available packet size.

We use our dataset to identify middleboxes that modify the MSS option in SYN segments. Figure 3.10 provides, for each VP (the horizontal axis), the proportion of paths (the vertical axis, in log-scale) where the MSS option has been modified. We see that a few VPs encountered at least one MSS modification on nearly all paths while, for the vast majority of VPs, the modification is observed in only a couple of paths. We decided to remove those VPs from our data set for further analyses, meaning that only 65 VPs were finally considered.

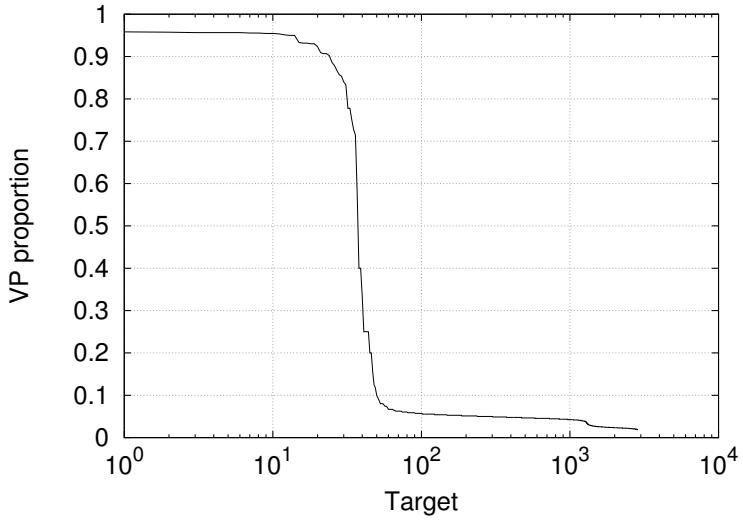


Figure 3.11: Targets proportion observing an MSS modification.

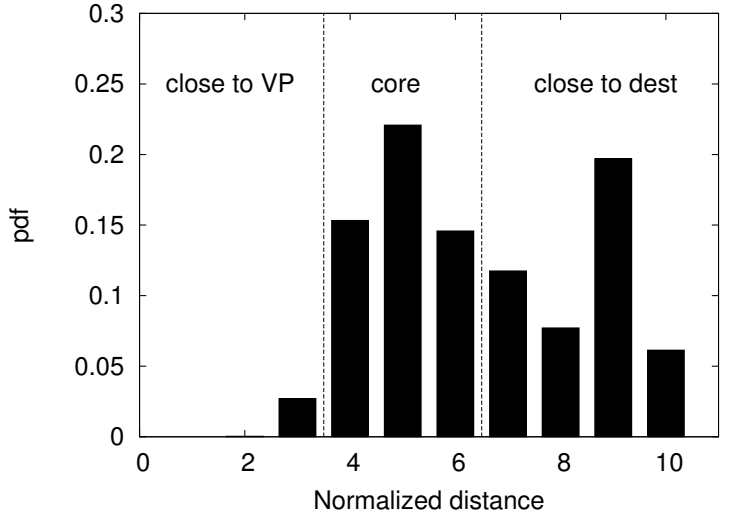


Figure 3.12: Location of middleboxes modifying the MSS.

Similarly to Figure 3.10, Figure 3.11 provides, for each target, the proportion of paths affected by an MSS modification. We see about ten targets that have a middlebox, probably their firewall or load balancer, always changing the MSS option. In the same fashion as the VPs that changed the MSS option, they also removed the Multipath TCP, MD5 and Window Scale options.

Figure 3.12 indicates where, in the network, the MSS option is modified. In the fashion of Figure 3.5, the distance from the VP has been normalized

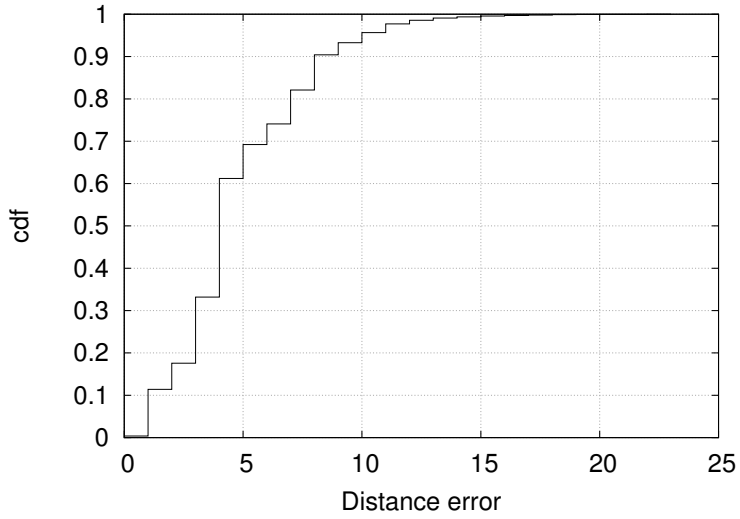


Figure 3.13: Location error of middleboxes modifying the MSS.

between 1 and 10, leading to three network regions (i.e., close to VP, core, and close to targets). As shown by Figure 3.12, `tracebox` can detect the MSS modification very close to the source (2.7% of the cases) while this detection mostly occurs in the network core (52% of the cases) due to MSS clamping and close to the destination (as a result of Figure 3.11).

Remember that this distance does not indicate precisely the location of the middlebox that modifies the MSS. Rather, it gives the position of the router that has returned a *Full ICMP* and, in this ICMP packet, the quoted TCP segment revealed a modification of the MSS field. Actually, the middlebox should be somewhere between this position and the previous router (on that path) that has also returned a *Full ICMP* (or the VP if it was the very first *Full ICMP* on that path).

Figure 3.13 refines our location of MSS modification by taking this aspect (i.e., the middlebox is somewhere on the path between the modification detection and the previous RFC1812-compliant router) into account. It gives thus an estimation of the middlebox location error. This error is simply obtained by subtracting the distance at which `tracebox` reveals the modification and the distance at which the previous RFC1812-compliant router was detected by `tracebox` on that path. Obviously, the lower the error, the more accurate the location given in Figure 3.12. On Figure 3.13, we see that in 61% of the cases, the location estimation error is below (or equal to) four hops. All errors above 13 hops, that represents the length of around 60% of the paths, are uncommon (less than 1% each).

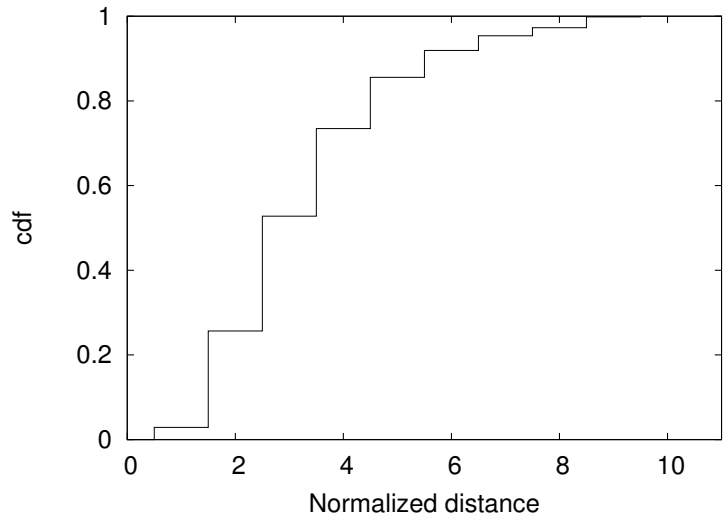


Figure 3.14: Location of middleboxes removing the MPTCP option.

3.4.5 Multipath TCP interference

Our last use case for `tracebox` concerns middleboxes that modify Multipath TCP options. We focused on the MP_CAPABLE option of MPTCP and generated SYN probes containing a valid option. Similarly to the MSS option, we do not at first sight expect this option to be modified or removed. However, the option is recent. The IANA assigned TCP option number 30 in march 2012. It is therefore possible that middleboxes configured to remove unknown options will consider this option as unknown and thus remove it.

Figure 3.14 indicates where the MPTCP option is modified. The horizontal axis corresponds to the distance, that has been normalized between 1 and 10, from the VP to the detection of the modification. The vertical axis shows the cumulative distribution of the modification. We can see that most of the middleboxes that remove the option are close to the source. Indeed, 70% of the modifications are detected upstream of the normalized fourth hop. This is due to the middleboxes that always modify several TCP options (see Section 3.4.1). These middleboxes correspond probably to a firewall configured to remove unknown TCP options. They remove the option by replacing it with TCP NOP options. Replacing by NOPs instead of a "clean" removal allows them to avoid any memory copy. Indeed, a "clean" removal of the option might require to move subsequent options while replacing by NOPs only requires to recompute the TCP checksum.

We analyzed the distribution of MPTCP option removal without the incriminated sources and observed that the modifications take place close to

the server. Compared to the MSS option, the modification never occurs in the core network but only at the edge of the network where firewall are placed (in front of clients and servers). Only a marginal fraction of the paths where impacted by the removal of the MP_CAPABLE option. Without the firewall at the source less than 1‰ of the paths where incriminated. More complex probing should be used to further evaluate the impact of middleboxes on MPTCP but this is a first indication that MPTCP can be used on today's Internet corroborating the conclusion of Honda et al. in [HNR+11].

3.5 Discussion

In Section 3.4, we showed that `tracebox` can provide a useful insight on known middleboxes interference. We believe that `tracebox` will also be very useful for network operators who have to debug strange networking problems involving middleboxes. While analyzing the data collected during our measurement campaign (see Section 3.4.1), we identified several strange middlebox behaviors that we briefly explain in this section. We also discuss how `tracebox` can be used to reveal the presence of proxies and *Network Address Translators* (NATs).

3.5.1 Unexpected interference

We performed some tests with `tracebox` to verify whether the recently proposed Multipath TCP [FRHB13] option could be safely used over the Internet. This is similar to the unknown option test performed by Honda et al. [HNR+11]. However, on the contrary to Honda et al., `tracebox` allows one to probe a large number of destinations. To our surprise, when running the tests, `tracebox` identified about ten Multipath TCP servers based on the TCP option they returned. One of those server, `www.baidu.com`, belongs to the top 5 Alexa. All these servers where located in China. A closer look at the returned options revealed that these servers (or their load balancers) simply echo the received unknown TCP option in the SYN+ACK. This is clearly an incorrect TCP implementation. Thanks to the fallback mechanism included in MPTCP, the connection is able to continue even if the server does not further reply with MPTCP options.

3.5.2 Proxy detection

`tracebox` can also be used to detect TCP proxies. To be able to detect a TCP proxy, `tracebox` must be able to send TCP segments that are intercepted by the proxy and other packets that are forwarded beyond it. HTTP

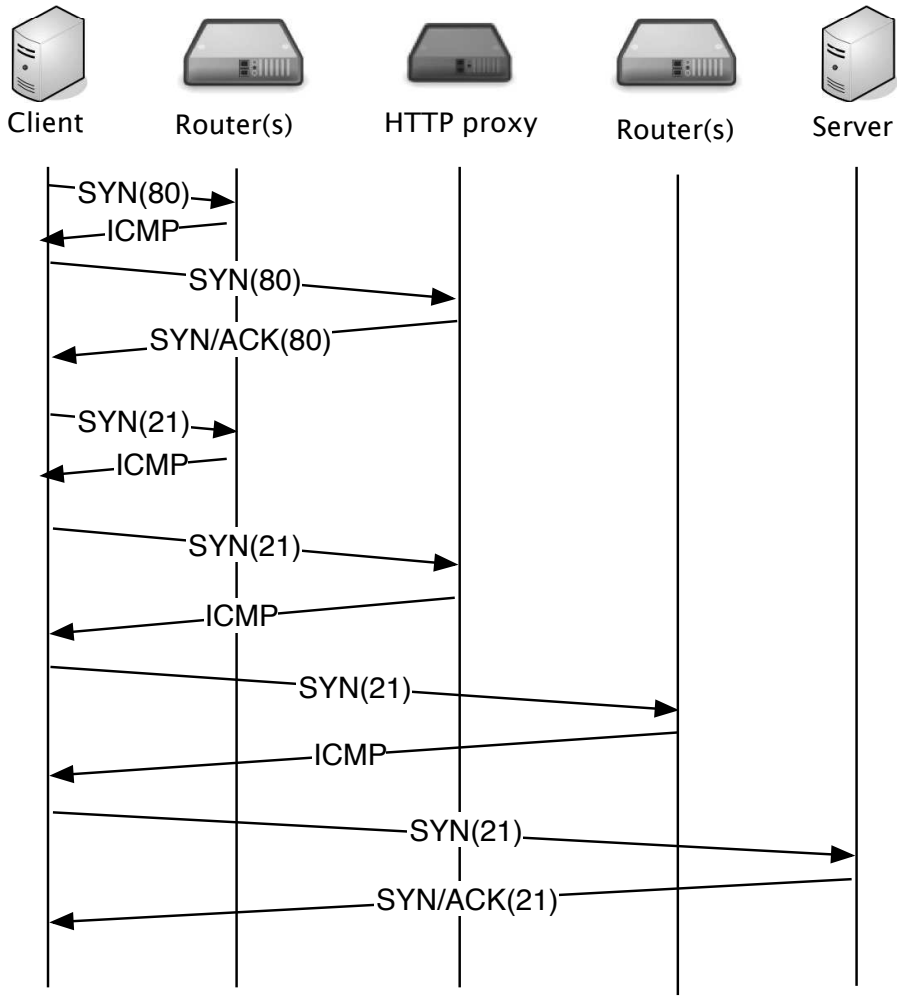


Figure 3.15: HTTP proxy detection example.

proxies are frequently used in cellular and enterprise networks [WQX+11]. Some of them are configured to transparently proxy all TCP connections on port 80. To test the ability of detecting proxies with `tracebox`, we used a script that sends a SYN probe to port 80 and, then, to port 21. Figure 3.15 shows how `tracebox` is used to detect such HTTP proxy. If there is an HTTP proxy on the path, it should intercept the SYN probe on port 80 while ignoring the SYN on port 21. We next analyze the ICMP messages returned. In the example of Figure 3.15 the source can easily determine that the paths are of different length (4 hops for port 21 and 2 hops for port 80) while the same set of routers have replied an ICMP message as to deal with

load balancing in the network where the two disjoint paths could have been taken while joining the same destination.

From our simple PlanetLab deployment, we identified two oddities. First, we found an HTTP proxy or more probably an IDS within a National Research Network (SUNET) as it only operated for a few destinations where the path for port 80 was shorter than for port 21. Second, and more disturbing, `tracebox` identified that several destinations where behind a proxy whose configuration, inferred from the returned ICMP messages, resulted in a forwarding loop for non-HTTP probes. We observed that the SYN probe on port 21, after reaching the supposed proxy, bounced from one router to the other in an endless loop (until the TLL expires) as `tracebox` received ICMP replies from one router to another alternatively.

3.5.3 NAT detection

NATs are probably the most widely deployed middleboxes. Detecting them by using `tracebox` would likely be useful for network operators. However, in addition to changing addresses and port numbers of the packets that they forward, NATs often also revert back the returned ICMP message and the quoted packet. This implies that, when inspecting the received ICMP message, `tracebox` would not be able to detect the modification.

This does not prevent `tracebox` from detecting many NATs. Indeed, most NATs implement *Application-level Gateways* (ALGs) [SH99] for protocols such as FTP. Such an ALG modifies the payload of forwarded packets that contain the `PORT` command on the `ftp-control` connection. `tracebox` can detect these ALGs by noting that they do not translate the quoted packet in the returned ICMP messages. This detection is written as a simple script (shown in Fig 3.16) that interacts with `tracebox`. It builds and sends a SYN for the FTP port number and, then, waits for the SYN+ACK. The script makes sure that the SYN+ACK is not handled by the TCP stack of the host by configuring the local firewall (using the filter functionality, shown at line 7, of `tracebox` that configures `iptables` on Linux and `ipfw` on Mac OS X). It then sends a valid segment with the `PORT` command and the encoded IP address and port number as payload. `tracebox` then compares the transmitted packet with the quoted packet returned inside an ICMP message by an RFC1812-compliant router and stores the modification applied to the packet. In Figure 3.16, the callback `cb` checks whether there has been a payload modification. If it is the case a message showing the approximate position of the ALG on the path is printed (see line 29).

```
1  -- NAT FTP detection
2  -- To run with: tracebox -s <script> <ftp_server>
3  -- Build the initial SYN (dest is passed to tracebox)
4  syn = IP / tcp{dst=21}
5  -- Avoid the host's stack to reply with a reset
6  fp = filter(syn)
7  synack = tracebox(syn)
8  if not synack then
9          print("Server did not reply...")
10         fp:close()
11         return
12 end
13 -- Check if SYN+ACK flags are present
14 if synack:tcp():getflags() ~= 18 then
15         print("Server does not seems to be a FTP server")
16         fp:close()
17         return
18 end
19 -- Build the PORT probe
20 ip_port = syn:source():gsub("%.", ",")
21 data = IP / tcp{src=syn:tcp():getsource(), dst=21,
22         seq=syn:tcp():getseq()+1,
23         ack=synack:tcp():getseq()+1, flags=16} /
24         raw('PORT '.. ip_port .. ',189,68\r\n')
25 -- Send probe and allow cb to be called for each reply
26 function cb(ttl, rip, pkt, reply, mods)
27         if mods and mods:__tostring():find("Raw") then
28                 print("There is a NAT before " .. rip)
29                 return 1
30         end
31 end
32 tracebox(data, {callback = "cb"})
33 fp:close()
```

Figure 3.16: Sample script to detect a NAT FTP.

3.6 Related work

Since the end of the nineties, the Internet topology discovery has been extensively studied [DF07, HRI+08]. In particular, `traceroute` [V. 89] has been used for revealing IP interfaces along the path between a source and a destination. Since then, `traceroute` has been extended in order to mitigate its intrinsic limitations. From simple extensions (i.e., the types of probes

sent [Tor01, LHH08]) to much more developed modifications. For instance, `traceroute` has been improved to face load balancing [ACO+06] or the reverse path [KBMA+10]. Its probing speed and efficiency has also been investigated [DRFC05, BBX10, BF13].

To the best of our knowledge, none of the available `traceroute` extensions allows one to reveal middlebox interference along real Internet paths as `tracebox` does.

Medina et al. [MAF04] report one of the first detailed analysis of the interactions between transport protocols and middleboxes. They rely on active probing with `tbit` and contact various web servers to detect whether *Explicit Congestion Notification* (ECN) [RFB01], IP options, and TCP options can be safely used.

3.7 Conclusion

Middleboxes are becoming more and more popular in various types of networks (enterprise, cellular network, etc.). Those middleboxes are supposed to be transparent to users. It has been shown that they frequently modify packets traversing them, sometimes making protocols' extension useless. Further, due to the lack of efficient and easy-to-use debugging tools, middleboxes are difficult to manage.

This is exactly what we tackled in this chapter by proposing, discussing, and evaluating `tracebox`. `tracebox` is a new extension to `traceroute` that allows one to reveal the presence of middleboxes along a path. It detects various types of packet modifications and can be used to locate where those modifications occur. `tracebox` is open-source and publicly available [Det13].

We deployed `tracebox` on the PlanetLab testbed and demonstrated its capabilities by discussing several use cases. It revealed interesting results such as the presence of firewalls that remove unknown TCP options, impacting therefore the deployment of MPTCP on the global Internet. In the next chapter, we try to tackle this deployment issue by deploying intermediate nodes on the Internet that supports MPTCP.

Chapter 4

Accelerating the deployment of MPTCP

4.1 Introduction

As we have seen in Chapters 1 and 2, smartphones have a motivation for using MPTCP as this would allow them to efficiently exploit their 3G and WiFi interfaces as well as support mobility. However, to reach the full benefit of MPTCP it must be supported on both the smartphones and the servers. This *chicken-and-egg* problem has been studied in [WLT+11]. Although the designers of the protocol took great care of avoiding interference with the various types of middleboxes [HNR+11, RPB+12], it is still expected that the deployment of MPTCP will take several years. It is also expected that clients will support MPTCP before servers. Indeed, *Apple Inc.* recently enabled MPTCP for a specific application in which they control the server-side [Bon13, app14]. They could enable it for other applications, if MPTCP was needed by other services.

In this chapter, we propose the utilization of protocol converters that we call *Multipath in the Middle Box* (MiMBox), to allow early adopters to benefit from MPTCP during its early deployment phase. MiMBoxes convert the MPTCP connections used by clients into regular TCP connections to allow clients to benefit from MPTCP even if it has not yet been deployed on servers. Economic studies show that such converters can play an important role in the deployment of a new protocol [SJGH10]. Figure 4.1 shows an example of how MiMBox can be used. The client has an interface to a WiFi and a 3G network. Thanks to MPTCP being used up to a MiMBox, the client benefits from MPTCP – even if the server does not support MPTCP. MiMBoxes can be placed in operator networks or on commodity servers in the cloud (e.g., as Network Virtualization Function [ETS12], etc.).

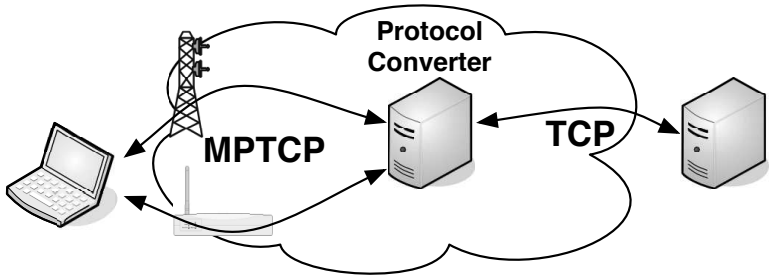


Figure 4.1: MiMBox translates MPTCP on the client-side to TCP on the server-side.

We saw in Chapter 3 that there are middleboxes that remove unknown TCP options on the Internet. Such middleboxes could prevent the use of MPTCP if present between the client and the MiMBox and thus it would void the interest of MiMBox. In this chapter, we make the hypothesis that such middlebox does not exist. However, as we control both end hosts, obfuscation techniques such as the TCP-over-UDP proposition [CGM13] that transform the TCP header into a UDP header in order to prevent middleboxes interfering with TCP. Other encapsulation techniques over UDP could also be used however TCP-over-UDP does not suffer from MTU issues as it does not requires additional bytes to be added to the original packet.

Our contribution in this chapter is the design, implementation and validation of a high-performance MiMBox, entirely implemented inside the Linux kernel. In addition to performing a protocol conversion, MiMBox provides the following features: *(i)* It can be *on-* or *off-path* to allow the client to explicitly contact MiMBoxes placed inside the cloud; *(ii)* If the server supports MPTCP, MiMBoxes can remove themselves from the connection. This reduces the load on MiMBox once the transition to MPTCP is progressing on the servers.

This chapter is organized as follows. Section 4.2 presents the design of MiMBox. Section 4.3 explains how this design is implemented inside the Linux kernel in order to achieve high performance. Section 4.4 discusses how to increase performance by taking advantage of multicore architectures. Section 4.5 presents a thorough evaluation of the solution and shows that MiMBox outperforms existing converters. Section 4.6 discusses the impact of the deployment of MPTCP on MiMBox. Finally, Sections 4.7 and 4.8 respectively discuss related work and conclude the chapter.

4.2 Network architecture

Deploying a new transport protocol is difficult and is often referred to as a chicken-and-egg problem. MPTCP, as every new protocol, suffers from this problem. Even by being backward compatible with regular TCP, neither servers nor clients have incentives to support it when the other end does not support it. To solve this adoption problem, we propose to deploy, across the Internet, MiMBoxes that transparently convert MPTCP from MPTCP-enabled clients to regular TCP used on legacy servers.

This section presents first how traffic can be redirected through the protocol converter. The the MPTCP-TCP conversion is described. Finally, the fallback mechanism when the server is MPTCP-capable is presented.

4.2.1 Transparent and Explicit

To allow the protocol conversion, TCP segments must be sent to a MiMBox. This can be achieved in two different modes: *Transparent* or *Explicit*.

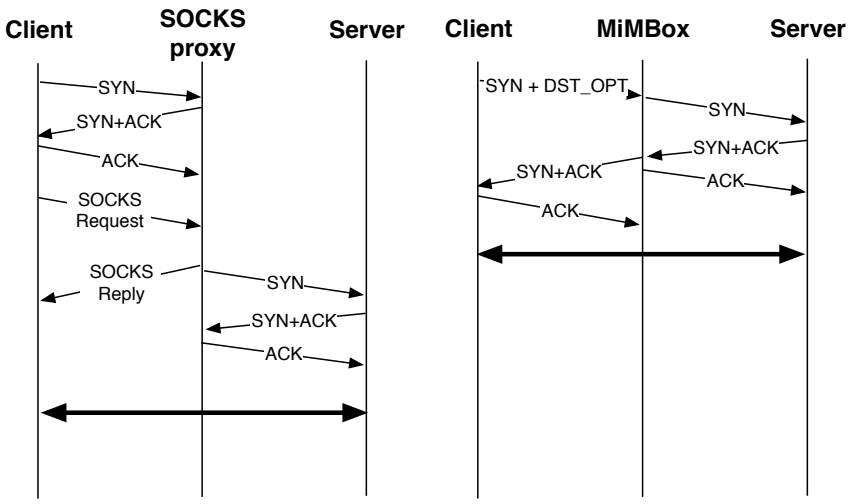
Transparent mode

In the transparent mode, the converter is invisible from the end hosts. The end host sends its traffic directly to the peer. It is the network that makes sure that the traffic passes through the converter.

To be transparent, MiMBox needs to either be on the path between the client and the server or a redirection mechanism must be in place in the network. The first solution is straight forward. If a MiMBox is on the path, it sees all traffic between the end hosts and can translate MPTCP to TCP. The second involves to setup a redirection mechanism in the network. This could be deployed at the border of an enterprise network or inside the xDSL routers present in home networks. They can both naturally intercept all traffic. The redirection can be based on a tunneling solution, where all traffic is explicitly sent to a MiMBox by the border gateway of the enterprise network (e.g., WCCP uses GRE tunnels [McL12], or a recent proposal by Sherry et.al [SHS+12]).

Explicit mode

With an explicit redirection, the client sends its traffic directly to a MiMBox in order to allow the latter to translate MPTCP to TCP. This is



(a) With a SOCKS proxy. (b) With a MiMBox.

Figure 4.2: MiMBox requires less round-trip before the end-to-end connection is established.

similar to the operation of an explicit proxy (e.g., an HTTP proxy) except that MiMBox is not limited to HTTP.

When using an explicit HTTP proxy, the client establishes a TCP connection to the proxy and modifies its HTTP requests to include the destination address. That way the proxy knows the original destination. Other solutions, like a SOCKS proxy require a specific application-level protocol to allow the client to indicate its desired destination (see Figure 4.2(a)). On top of requiring that the applications or Operating Systems support the proxy mechanism it also delays the connection establishment. This can be a burden for the use of this kind of redirection mechanisms.

MiMBox does not modify the application layer. Instead we propose a new TCP option, that we call Dst_Opt, to allow the client to specify/indicate the server address. The Dst_Opt provides the server's IP address to the MiMBox. Figures 4.2(b) and 4.3 show how the establishment of new connections is performed via a MiMBox. When establishing a new connection the client places the Dst_Opt inside the SYN segment and the destination address for this connection is the address of the MiMBox. This allows the latter to forward the connection establishment to the server by rewriting the segment's IP addresses. By using its own IP address, all reply segments will be sent via the MiMBox. The Dst_Opt is added by the MPTCP/TCP stack and is thus transparent for the application.

Note that the use of explicit redirection allows to easily scale up MiM-

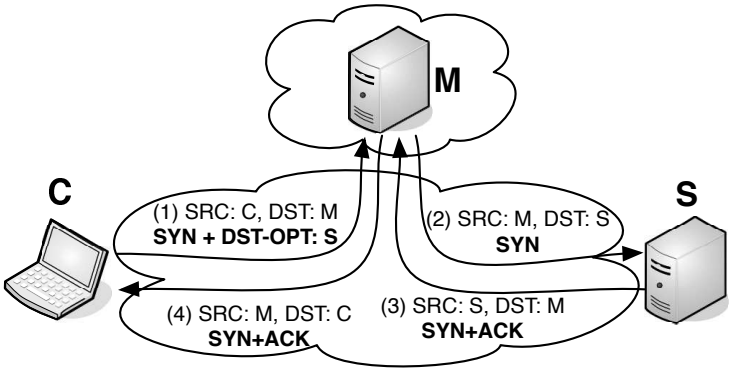


Figure 4.3: Explicit redirection of connections establishment through the protocol converter using the DST_OPT TCP option.

Box. We can configure the clients to perform DNS queries to retrieve the address of a MiMBox and therefore perform DNS load balancing [Bri95]. This is a regular way of distributing the load across multiple devices.

To ensure that the converter sees the bidirectional flow in the explicit mode, it must act as a NAT. By using its own IP address[1] all the segments will be sent via the converter. In the rest of this chapter we focus on the explicit mode, however all principles apply similarly to the transparent mode.

4.2.2 Protocol conversion

The purpose of the protocol converter is to translate MPTCP connections to TCP connections. The conversion can be divided in two operations: *(i)* detect MPTCP capabilities of the end hosts; *(ii)* translate MPTCP segments to TCP and vice versa.

To detect whether the server supports MPTCP, the converter parses the SYN+ACK from the server as it sees the connection establishment. If the server supports MPTCP, the SYN+ACK includes the MP_CAPABLE option. If the SYN+ACK does not include this option, the protocol converter starts converting the TCP segments from the server to MPTCP segments for the client. If the server does not support MPTCP, MiMBox will include a MP_CAPABLE option in the SYN+ACK sent back to the client such that it believes that it can use MPTCP for the rest of the connection.

[1] MiMBoxes can be configured to use a block of IP addresses.

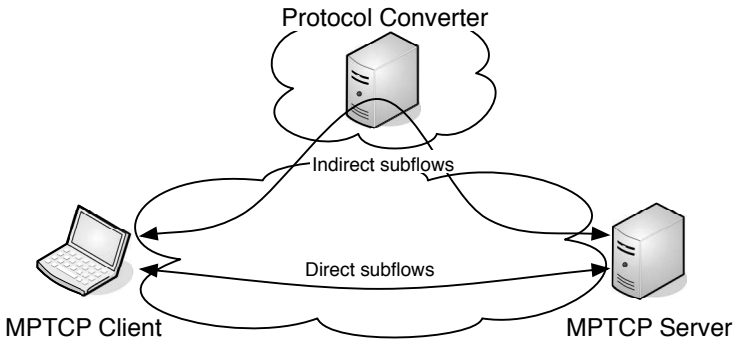


Figure 4.4: MiMBox allows the client to establish direct subflows to the server if both support MPTCP.

The operations performed by the MiMBox to translate data segments can be viewed as a pipe, channeling segments from TCP to MPTCP and vice versa. Incoming segments on the MPTCP side contain MPTCP options inside the TCP header. MiMBox has to handle the options (e.g., new subflow establishment, etc.) and strip these options before forwarding them. MPTCP uses a different sequence number space than the TCP sequence numbers [RPB+12]. When forwarding a segment, MiMBox translates the MPTCP-level sequence numbers to the TCP sequence numbers on the server-side and vice versa. Further, as the TCP/IP header is modified, MiMBox updates the TCP checksum.

The converter can also forward segments from a TCP client to an MPTCP-enabled server, e.g., configured as a load balancer. In this case, it needs to add the MP_CAPABLE option when forwarding the SYN. We do not further evaluate this mechanism in this chapter as all features apply similarly.

4.2.3 Fallback

When both end hosts support MPTCP, the converter does not need to aggregate the segments from the different subflows. In this case, after the three-way handshake, the protocol converter acts as a simple forwarder between end hosts. In explicit mode, it needs to change the IP addresses. As the converter does not need to reorder the segments, the client and the server can create additional subflows as shown on Figure 4.4, without passing through the protocol converter. By creating direct subflows between the client and the server the communication can be speed up, as the round-trip time is reduced.

To allow these direct subflows between client and server, the protocol converter must announce this capability to the end hosts. This is achieved thanks to the *anchor* flag in the MP_CAPABLE option [HK12]. The anchor flag is added by the proxy when it forwards the SYN+ACK to the client. Upon reception of a SYN+ACK with the anchor flag set, the client knows that it can establish a direct subflow to the original server's address.

We further evaluate in Section 4.6 the impact of MPTCP-enabled servers on protocol converters as well as further improvements to the fallback solution.

4.3 In-kernel protocol converter

A MiMBox could be implemented as a user-space application. Existing HTTP proxies, such as Squid [squ12] and HAProxy [hap12] are applications running in user space. This simplifies the development but may affect performance. First, these proxies are limited to specific applications and services. Each of these services runs on a specific port. Additionally, the application needs to include a redirection mechanism to allow the *explicit mode* of MiMBox. MiMBox is application agnostic and does not require any application change.

To achieve high performance, the following goals are important:

Avoid Memory Allocation/Copy
: The memory bus is a major limiting factor of the overall system performance. The gap between the processor and memory performance is important and still increasing [JNW10]. To overcome the memory access bottleneck it is preferred to avoid memory allocations and copies.

Minimize Context Switching
: Limiting the number of context switches is important to achieve high performance. Context switching consumes CPU cycles that could have been used for other resources and it has an effect on the space locality of the data in the caches if processes are moved from one CPU to another.

To achieve these two goals MiMBox is implemented as a kernel module. This allows to avoid memory copy between user space and kernel space as well as to react quickly when packets arrive in the TCP/IP stack. Moreover, running in kernel space reduces the footprint. We can run MiMBox in a virtual machine with KVM [KKL+07] using only a 10MB kernel and a 800KB initramfs. The latter is used as the root filesystem and runs a dummy application that sleeps indefinitely. The module receives all incoming packets

and performs two operations: *(i)* handling connection establishments from the clients and *(ii)* forwarding packets.

In this section we present our design that consists of around 3K lines of kernel code. First, we describe our main design choices. Second, we present how MiMBox handles connection establishments. Then, we present how segments are forwarded from one side to the other. Finally, we present how the checksum is updated.

4.3.1 Design choices

One major design choice of MiMBox is to use the already existing MPTCP-stack of Linux. MiMBox could have been implemented using solutions such as *netmap* [Riz12] that allows a user-space application to interact directly with the NIC without going through the host's stack. While this brings high benefit for simple application such as switching, it still requires to implement a complete TCP/IP stack in user space to support MiMBox. As there does not exist such stack, it is unclear whether such implementation would achieve better performances. Furthermore, from our experience of implementing MPTCP in the Linux kernel, we know that implementing a new stack from scratch takes time and might be bug prone.

As we reuse the Linux implementation, we decided to terminate both connections. Therefore, as MPTCP creates multiple subflows, segments can arrive independently on each of these subflows. MiMBox therefore reorders the segments so that they form an in-order sequence of packets. Finally, MiMBox sends this sequence to the TCP side. For segments coming from the TCP side, MiMBox distributes the segments among the subflows by using MPTCP's scheduling algorithm (see Section 1.3.3).

4.3.2 Connection establishment

MiMBox receives connection establishments from the client and forwards it to the server. There are two cases: either the server is MPTCP-capable or not. We present here the operations done inside the kernel to forward the segments during the three-way handshake.

When a MiMBox receives a SYN segment, it creates a lightweight state containing the addresses and ports used as well as some TCP and MPTCP informations. This consumes around 100 bytes which is small compared to the ± 2K bytes memory footprint of a fully functional TCP socket.

In explicit mode the protocol converter uses the same IP address on the server-side. When receiving a connection establishment, it must select the

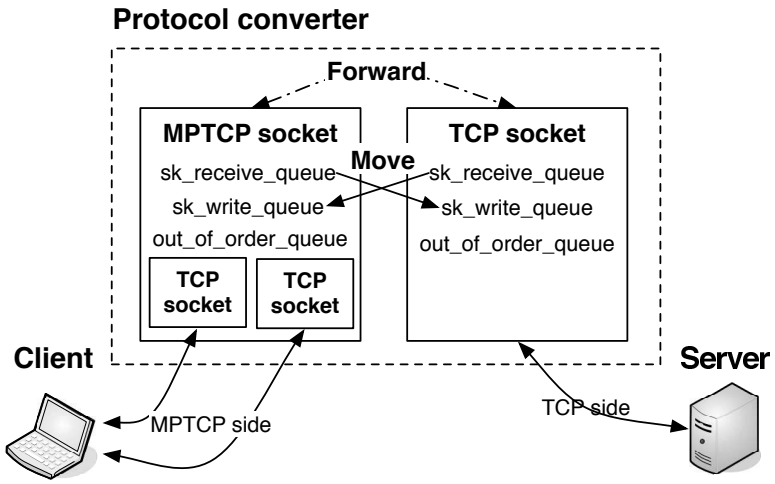


Figure 4.5: MiMBox maintains fully-functional sockets and forwards in-order segments from one side to the other.

source port on the server-side so that it does not collide with pre-existing states or established sockets.

The state is stored in two distinct hash tables so that a lookup is possible for segments from both the client-side (SYN retransmission) and the server-side. When receiving a SYN+ACK from the server, a lookup is performed and the state is retrieved. Further operations depend on the two cases presented in Section 4.2.3. If the server is MPTCP-enabled the lightweight state is maintained so that data segments are forwarded by modifying their header. We focus in the rest of this section on the server is TCP-only case.

When the server only supports TCP, the MiMBox converts the TCP segments from the server-side to MPTCP segments on the client-side. The converter terminates the three-way handshake of the TCP and MPTCP connections. It creates two fully-functional sockets: the TCP socket and the MPTCP socket. Figure 4.5 shows the behavior of the protocol converter in this scenario. The following section explains how segments are forwarded from one socket to the other.

4.3.3 Forwarding segments

To understand how MiMBox can achieve high performance, one must understand how network packets are handled inside the kernel. The Linux kernel uses special buffers, called `sk_buff`s [Cox96], to handle network packets. When a segment arrives at the NIC a `sk_buff` is allocated. The buffer

```
1: procedure FORWARD(receive_queue, write_queue)
2:     while !ISEMPTY(receive_queue) && CANWRITE(write_queue)
   do
3:         skbuff ← DEQUEUE(receive_queue)
4:         RECOMPUTECHECKSUM(skbuff)
5:         REWRITESEQUENCENUMBERS(skbuff)
6:         ENQUEUE(write_queue, skbuff)
7:     end while
8: end procedure

9: procedure RECOMPUTECHECKSUM(skbuff)
10:    Recompute the payload's checksum
11:    skbuff→csum = TCP Checksum - CHECKSUM(TCP Header) -
   CHECKSUM(Pseudo Header)
12: end procedure
```

Algoritm 4.1: The Forwarding Procedure.

then traverses the TCP/IP stack and, for a TCP connection, is finally stored in the receive queue of its corresponding socket waiting for the application to copy the payload into its own buffer. `sk_buff`s are also allocated by the kernel when the application wants to send data. In this case, the content of the user-space buffer of the application is copied inside the `sk_buff`'s data part.

Figure 4.5 shows the operations performed to forward segments from one socket to the other: the segments are moved from the receive queue (`sk_receive_queue`) from one socket to the write queue (`sk_write_queue`) of the other. A user space application that would move data from one socket to another using the `read()` and `write()` system calls would cause two buffer allocations and two memory copies. To achieve high performance, MiMBox limits the number of allocations and avoids costly copy operations. Indeed, the forwarding mechanism is only triggered when an event occurs on the socket: either data can be read from the receive queue or space is available in the write queue. The operations performed consist of moving `sk_buff`s from the receive queue of one socket to the send queue of the other socket and vice versa. This is done by modifying pointer references as well as updating sequence numbers and checksum. The cost of these operations is therefore minimized.

Algorithm 4.1 shows the pseudocode of the forwarding procedure. When an event occurs on the socket pair, we first iterate on the receive queue of the socket to retrieve in-order segments. For each `sk_buff` present in the receive queue, we move them to the other socket's write queue after modifying

the sequence numbers. If no space is available in the write queue, we ensure that our forwarding mechanism is called when space becomes available. This ensures that no memory allocation and copying is performed during the forwarding procedure, i.e., `sk_buff`s are moved as is.

Sequence numbers

The sequencing of data in MPTCP differs from TCP: MPTCP uses its own data sequence numbers. When forwarding segments from the MPTCP to the TCP socket, the data sequence number must therefore be translated to the TCP sequence number and vice versa.

A sequence number is translated using the formula: $seq_{new} = seq_{rcv} - isn_{rcv} + isn_{snt}$. The sequence number of the segment being forwarded is given by seq_{rcv}. Each segment has a relative position inside the data stream, given by the operation $seq_{rcv} - isn_{rcv}$ (isn_{rcv} is the initial sequence number of the receiving socket). This relative position must be added to the initial sequence number of the socket where the segment is sent on (isn_{snt}). This latter operation adapts the segment's sequence number to the sequence number of the sending socket, where the data is sent. Of course, as sequence numbers are based on a 32-bit unsigned integer, the resulting seq_{new} must be also limited to 32 bits.

Timestamps

TCP timestamps is a TCP extension defined in RFC1323 [JBB92]. Originally defined to be used in the *Protection Against Wrapped Sequence numbers* (PAWS) algorithm to break the tie when detecting segments retransmission. It is now also used in the mechanism that allows to estimate the *Retransmission Timeout* (RTO) accurately. The protocol converter must therefore take care of sending valid timestamps on both sides. The converter forwards the SYN as is, in order to be able to fallback if the server is MPTCP-capable (see Section 4.2.3). The server therefore sees the timestamp from the source. End hosts typically compute the timestamp based on the *jiffies* of the operating system. This value differs from one machine to another. As the converter is forwarding from the MPTCP-side to the TCP-side, the timestamps must comply with the one provided in the original SYN. This is possible by storing the difference between the timestamp in the SYN segment and the local timestamp inside the converter when forwarding this SYN. The converter must make sure that the timestamps in the segments evolve based on the original timestamp in the SYN segment, by applying the aforementioned difference, as to emulate the timestamps generated by the end hosts.

4.3.4 Checksum

As explained in Section 1.3.3, MPTCP includes a checksum in the DSS option to protect against payload modifying middleboxes. When forwarding traffic from the TCP-side to the MPTCP connection, this checksum must be calculated. The checksum is based on the payload and the data sequence number. Calculating the TCP checksum is usually offloaded to the NIC. Unfortunately, it is currently impossible to offload the computation of the DSS option's checksum as there does not exist a NIC that supports this type of checksum.

A naive approach to compute the checksum would be to recompute the complete payload's checksum. However, this solution consumes a large number of CPU cycles as the payload's size can be important (9k bytes with jumbo frames). We propose an iterative solution, where the checksum is recomputed based on the TCP checksum present in the segments (see procedure RecomputeChecksum of Algorithm 4.1). Indeed, as the segment is coming in from the TCP stack, it is known that the TCP checksum inside the TCP header is correct. The TCP checksum is computed from the payload's checksum, the TCP header and the pseudo IP header. To efficiently recompute the payload's checksum, we substitute from the TCP checksum, the checksum over the TCP header and the pseudo header. Using this approach results in fewer CPU cycles consumed as the checksum is only computed over at most 72 bytes with IPv4 and 100 bytes with IPv6, instead of calculating the checksum over the whole payload.

4.4 Multicore architectures

Multicore systems are becoming ubiquitous. It is thereby desirable to allow to spread the load over the available cores in order to improve the overall performance. The protocol converter can also benefit from multicore architectures, as it allows forwarding several streams at the same time.

Using a multicore architecture means that the converter has to handle the parallel processing of packets and thus the synchronization between the sockets. Furthermore, multicore processing is known to raise a challenge in terms of lock contention and cache misses [VF07, PSZM12, WDC11]. The following sections present how the protocol converter solves these two challenges.

4.4.1 Locking

The protocol converter, in addition to forward segments, needs to handle the locking. Indeed, two packets may arrive at the converter in the TCP and the MPTCP sockets at the same time. To understand how our implementation handles the locking we first give an overview of how the Linux kernel handles the locking in a normal operational mode. The Linux kernel locking architecture of the TCP stack involves two cases: *(i)* The NIC receives a packet and *(ii)* the application sends/reads data to/from the socket.

When the NIC receives a packet, it triggers an interruption that is handled in soft interrupt context. This execution context does not allow any other process or other soft interrupt to execute on the processor core during the interruption. Upon the reception of a packet, the Linux kernel takes a lock on the socket inside this soft interrupt context. Therefore, if an application tries to read or write from the socket while another CPU core is handling a packet reception on this socket, the application blocks until this CPU core has released the lock.

The behavior is different if the application holds the socket lock to read data from the socket while a packet arrives on the NIC. While handling the reception, taking the lock is impossible as the application already holds it. The soft interrupt would need to sleep until the application releases the lock. However, sleeping is prohibited inside soft interrupts. The packet processing is thus deferred to the application by appending the packet to the *backlog queue*.

The protocol converter involves a different handling of locks, because forwarding needs to be done from one socket to the other. Thus, both sockets must be locked before entering the forwarding procedure. A simple solution would be to define a shared lock among the MPTCP and the TCP sockets and simply take this lock upon reception of a packet. This simple approach however has two drawbacks. First, the shared lock would also be taken when the socket is only receiving an ACK. An ACK will not be forwarded from the MPTCP to the TCP socket as both sockets handle data acknowledgments independently. Second, we want to minimize the modifications to the regular TCP stack. We would need to change the locking architecture of the Linux kernel to implement a shared lock.

Our approach to locking inside the MiMBox does not require changing the Linux kernel locking architecture. We still define a shared lock, but this shared lock is only used if we need to forward data from MPTCP to TCP. Our locking architecture involves the following steps when a packet is received on the TCP side, converted, and forwarded to the MPTCP side:

1. Take the TCP socket lock upon receiving a packet from the NIC;
2. The packet is received through the regular TCP stack and will be forwarded by the MiMBox;
3. The converter releases the lock of the TCP socket;
4. The shared lock is taken;
5. MiMBox takes the socket lock of both the TCP and the MPTCP socket.

Steps 1 and 2 correspond to the default Linux behavior. At step 3, our implementation redirects the packet handling to the protocol converter. The first lock to take must be the shared lock. However, as the TCP stack has already locked the TCP socket, we need to release this one before. Indeed, we need to ensure that all locks are taken in the same order as to avoid any deadlock. If the other socket has data to forward, it must release its lock before taking the shared lock. This ensures that only one socket is in the critical region, i.e., the forwarding procedure, at the same time. After acquiring the shared lock, the lock from both sockets must be taken to block new data that may arrive from the NIC on either socket. After the forwarding procedure, the lock state is reverted to the one before, namely the shared lock is released as well as the lock from the socket on which we forwarded the data.

4.4.2 Flow-to-core affinity

It is known that flow-to-core affinity [SKT96] improves the performance of network packet processing. Flow-to-core affinity has multiple advantages. First, without flow-to-core affinity, if packets of a TCP connection are received on different cores, they may get reordered inside the receive code of the OS's TCP/IP stack [VF07]. This reordering can reduce the TCP performance as it causes reordering in the data which can lead to duplicate acknowledgements that can be interpreted as a loss on the sender side. Second, flow-to-core affinity reduces the number of cache misses and mitigates the effect of lock contention. Several authors have shown the benefits of optimizing packet handling by taking into consideration the core on which the packet is being received [PSZM12, WDC11].

For the protocol converter the flow-to-core affinity must be taken one step further. Indeed, we do not have a single TCP connection that should be handled on a single core, but we have a TCP socket and an MPTCP socket (together with all its TCP subflows) that should be handled on the same CPU core. Usually, flow-to-core affinity is handled by the NIC. However,

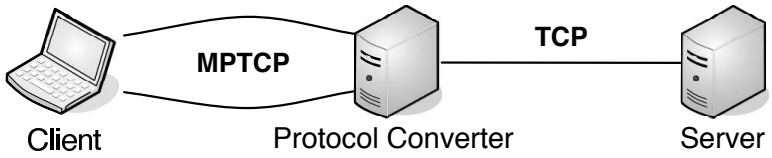


Figure 4.6: The setup used throughout the experiments.

adapting the NIC to support flow-to-core affinity for the protocol converter would involve major changes in the NIC's firmware.

In our implementation we take a different approach by using the *Receive-Flow-Steering* (RFS) framework of the Linux kernel [Her10]. RFS allows the operating system to specify on which CPU core packets from a certain flow should be received. Originally, RFS was designed to steer all packets from a TCP connection to the CPU core where the application using this TCP connection is running. This reduces the number of cache misses as the data does not need to be transferred from one CPU core to another. In MiMBox we enforce all packets of the TCP socket, the MPTCP socket and its TCP subflows to be received on the same CPU core. The performance improvement of this approach is significant and is described in Section 4.5.

4.5 Evaluation

In this section we first describe the experimental setup and then evaluate the performance of MiMBox and compare it with user-space solutions.

4.5.1 Hardware Setup

We run experiments on three hosts connected as described in Figure 4.6. They are all Intel Xeon X3440 running @2.53GHz and have 8GB RAM. Each of these machines has a quad-port NetXen/Qlogic NX3031 1Gbit and a dual-port Intel 82599 10Gbit Ethernet networking cards. Though we have numerous Ethernet interfaces, we only use 2 of them at the same time[2], either in a 1Gbit or in a 10Gbit setup. Both Ethernet cards support hardware offloading.

Each host runs the v0.87 release of MPTCP[3]. On top of that, the converter runs our MiMBox implementation based on this same MPTCP

[2]The bottleneck is the link between the converter and the server which consists of a single link.

[3]Freely available at http://multipath-tcp.org

implementation. Depending on the evaluation, the client and the server run different pieces of software, they use either Weighttp [wei13] and Apache or Netperf [Jon13]. The protocol converter is where the evaluation takes place. It is running either MiMBox, Squid, HAProxy, a custom application-level converter or acts as an IP router. Squid and HAProxy are two well known HTTP proxies which are often used as caches or HTTP load balancers. They both terminate the client connections and start a new connection towards the server.

4.5.2 Goodput

Inserting protocol converters on the path of the data stream between the client and the server may reduce the throughput, i.e., the rate at which the end hosts can send/receive data over the network. To measure the impact on the application-level throughput (called goodput) we use Netperf. Netperf allows to send data from the client to the server at the highest possible speed up to the network capacity. Netperf then reports the goodput it experienced during the experiment.

Our baseline to compare the performance is the setup where the converter is acting as a regular IP router. As the router does not perform any protocol conversion, we enable MPTCP on the server-side for this scenario and thus obtain the best possible performance.

We implemented two application-layer protocol converters to compare our kernel-space implementation with a user-space one. These converters open a socket using the Netperf port on the client-side and listen for incoming connections. When a connection is established, the converter opens a new connection towards the server using the same port. The first application uses the `read()` and `write()` system calls to forward the data received on one socket to the other. The second application uses `splice()` [MB00]. `splice()` is a Linux specific system call that allows to move data from one file descriptor to another without having to copy the data from kernel to user space. Using `splice()` from one TCP socket file descriptor to another can be seen as forwarding at the TCP layer. Splicing is often employed in web proxies [MB00] where, after inspecting some part of a connection in the application, it can make the rest of the connection fall back to TCP forwarding for optimal processing.

It must be noted that these application-level protocol converters are difficult to realize in practice. They must listen explicitly on a port and so are not application agnostic. Furthermore, current applications do not allow to specify the real server's destination IP address, they must modify the application data stream.

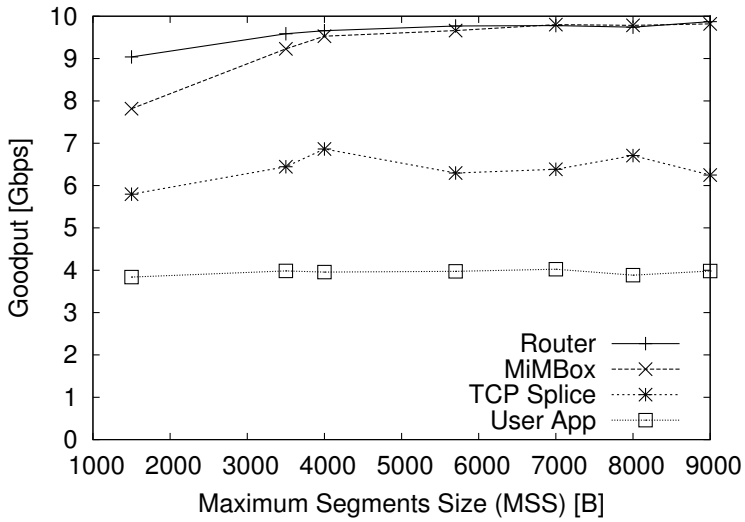


Figure 4.7: MiMBox always outperforms application-level solutions.

We evaluate the impact of the *Maximum Segment Size* (MSS) on the goodput. Figure 4.7 show the average goodput over 10 measurements for different MSS values. We can observe that MiMBox achieves better performance, very close to simple IP forwarding. Overall, with a small MSS the maximum goodput, i.e., 10 Gbps, is not achieved with any solution. The lower the MSS, the higher the number of segments that must be transmitted to achieve full goodput. The number of segments a device can generate/receive is function of the number of interruptions that it can handle. The number of segments generated/received is therefore fixed. Increasing the MSS size thus increases the goodput, as a larger amount of data is transmitted per segment.

The performance difference between MiMBox and its user-space counterparts can be explained by the fact that the latter ones are CPU limited due to memory allocation and copy. `splice()`'s performance is mainly impacted by the memory allocation. When performing `splice()`, pages are moved, however `splice()` causes a new `sk_buff` allocation while this is not required in MiMBox which moves `sk_buff`s as-is. We can observe two spikes when using `splice()`, these are located around MSS values of 4KB and 8KB. These are the same value as the pagesize used on the devices. We can see that after 4KB, the kernel has to allocate two pages instead of one to carry the data and thus it is more costly and impacts the goodput. The same occurs at 8KB but in this case it allocates 3 pages.

The curves of Figure 4.7 can be decomposed in two parts, depending on their slope. First, on the left side with a high slope, the end hosts are not able to generate traffic at the maximum speed (e.g., with a MSS smaller than

4KB for the router). Second, on the right side, when the slope decreases and is close to null, the limitation is either due to the network capacity (Router and MiMBox) or to the converter software that uses all CPU cycles available to forward (TCP Splice and User App). For the router, the limitation is the bandwidth available in the network while the others are limited by the software except for MiMBox that achieves the maximum goodput for high MSS values.

The difference of performance between routing and the other solutions is due to the incoming segments/data stream that needs to pass twice through the TCP stack. Once for the reception of the data, once for sending the data back on the other side of the converter. The custom application-layer protocol converter's performance is almost exactly half of the router's performance as the data is recomposed at the application layer and thus a complete transition from user space down to the NIC via the kernel space is done twice. MiMBox still needs two complete TCP stacks, however without the need to transition from user space to kernel space. Moreover, the difference between the application and MiMBox is also due to the context switches applications undergo and due to cache misses that are higher. We measured that the application suffers from 200 % more cache misses than MiMBox. These cache misses are mostly (65 %) due to the memory copy operations from the kernel to the user space and vice versa.

We also evaluate the impact of having pre-established connections between the client and the server (passing by a MiMBox) and how these connections affect the performance. We pre-charge the application converters and MiMBox with up to 20000 connections that are not sending data in order to evaluate the cost of filling up the hash tables of the kernel. We do not observe any impact on the performance.

4.5.3 Forwarding delay

One important performance factor is the forwarding delay introduced by MiMBox. To validate our design we measure the forwarding delay which is the time spent by each packet inside the converter. We use a custom application that sends blocks of 1300 bytes of data at 20 Kbps. We capture all packets entering and leaving the converter with `tcpdump` into a *ramdisk*. `tcpdump` stores, together with the packet, a timestamp of the moment the packet enters/leaves the host. This timestamp allows us to measure the time spent inside the host by each packet forwarded by the converter. We also ensure that a route cache exists prior to measuring the delay to avoid influencing the SYN and SYN+ACK delays.

As in the previous section, the baseline is the performance achieved us-

	SYN	SYN+ACK	Data
Router	5.2 ± 0.1 µs	5.2 ± 0.1 µs	5.3 ± 0.1 µs
`read()`/`write()` App.	143.4 ± 11.4 µs	N/A	29 ± 0.3 µs
MPTCP-TCP	17.5 ± 1.2 µs	41.6 ± 2 µs	16 ± 0.1 µs
MPTCP-MPTCP	14.2 ± 1.2 µs	12.7 ± 0.9 µs	9 ± 0.1 µs

Table 4.1: MiMBox introduces a moderate per-packet delay – application-level solutions are much worse.

ing the converter as a regular IP router. We profile MiMBox with a TCP and MPTCP-enabled server as well as the previously described `read()`/`write()` application. If the server is MPTCP-enabled, MiMBox will go into fallback mode (see Section 4.2.3). MiMBox works in this mode similar to a NAT. Table 4.1 shows the per segment forwarding delay introduced by each of the solutions averaged over 100 measures as well as the 95% confidence interval.

A first observation, is that the difference between the application and MiMBox is large for SYN packets. The poor performance of the `read()`/`write()` application is because the `accept()` call only returns when the ACK is received on the client-side of the converter. Therefore, the connection on the server-side is only established after the three-way handshake is completed on the client-side, resulting in a higher delay. Measuring the SYN+ACK delay is impossible since the SYN+ACK on the client-side is sent before the SYN+ACK from the server-side.

The difference in the SYN forwarding delay between MiMBox and the router is due to the additional operations performed by MiMBox. While the former needs to match to a route cache entry, the latter needs to create a state (see Section 4.3.2), change the IP addresses, verify the port numbers to avoid 5-tuple collisions and finally recompute the TCP checksum before forwarding the SYN to its final destination.

When the server replies with the SYN+ACK, the forwarding through MiMBox depends upon the presence of the MP_CAPABLE option in the SYN+ACK. An MPTCP-enabled server will add this option in the SYN+ACK. In this case, MiMBox is in fallback mode and uses the lightweight state to continue forwarding the data segments. If the server only supports TCP, the MiMBox must allocate fully functional sockets as well as the MPTCP structure for the client-side. It cannot forward the SYN+ACK before the creation of these data structures. This additional delay causes MiMBox to be slower in this case.

After the three-way handshake, the forwarding delay for segments containing data is smaller than for SYNs and SYN+ACKs as there are fewer op-

erations to perform. If the server supports MPTCP (MPTCP-MPTCP), the main operation is to lookup for a lightweight socket and rewrite the addresses and ports as well as recompute the TCP checksum. If the server only supports TCP (MPTCP-TCP), the segments pass through the TCP/MPTCP stack to be forwarded. Traversing inside the two stacks introduces this additional delay.

The `read()`/`write()` application gives worse results when forwarding data as it consumes many CPU cycles transitioning from kernel space to user space.

4.5.4 Workload

In the previous sections we evaluated the performance of a single connection through MiMBox. We now evaluate the cost of supporting network-heavy applications on the global performance. We use weighttp running 40 parallel clients sending HTTP requests for varying file sizes. We measure the number of requests per second that the client is able to perform.

As a baseline we again use the kernel-based router. We also profile existing HTTP proxies: HAProxy and Squid. These proxies terminate the MPTCP connections on the client-side and start a new TCP connection on the server-side. Their caching mechanisms are disabled as we want to measure pure forwarding performance. These proxies are application specific but still act similar to MiMBox at the data level. They forward the HTTP requests and the response back to the client. We configured HAProxy to use `splice()`. For this the proxy performs a `read()` of the client requests so that it can lookup the destination server in the HTTP header and establish a new connection. When the reply from the server comes back, HAProxy uses `splice()` to forward the data to the client. Compared to Squid, HAProxy can use multiple threads to handle incoming requests. As our test server has 8 cores, we configured HAProxy to use either 1 or 8 threads. Using multiple threads allows to increase the CPU cycles available but also increases the number of cache misses.

Figure 4.8 shows the average number of requests per second supported (in log-scale) over 10 measurements for different request sizes. The figure shows that MiMBox outperforms both HAProxy and Squid. For a request of a 1KB file respectively MiMBox, multi-threaded HAProxy, mono-threaded HAProxy and Squid are able to sustain around 22k, 18k, 9.5k and 2.5k requests per second. The result can be explained by the fact that MiMBox is quickly able to fully utilize the available bandwidth as well as optimize the three-way handshake. Indeed, the HTTP proxies have to wait for a HTTP GET request from the client to identify the server and establish the server-

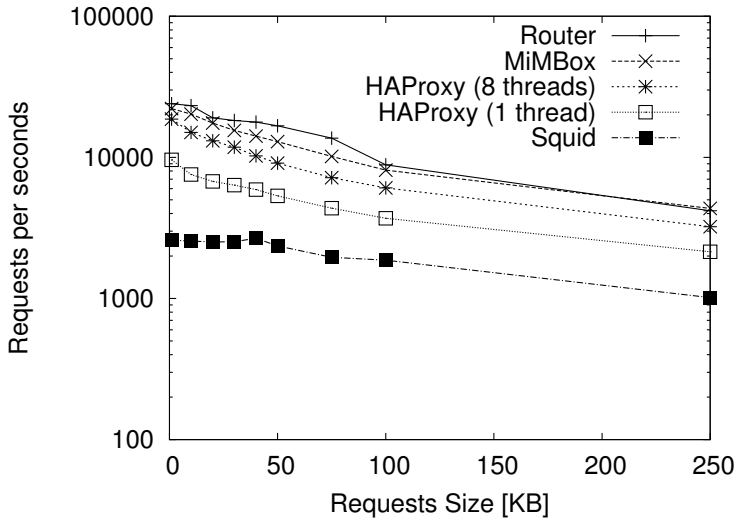


Figure 4.8: MiMBox supports a large number of HTTP clients – close to the performance of an IP router.

side connection. The HTTP proxy also has to parse the server address in the HTTP header. Figure 4.8 also shows that MiMBox sustains a similar number of requests per second as plain IP forwarding. For large file sizes the cost of establishing new connections is negligible. Nevertheless, in this case MiMBox still outperforms the user-space proxies and the limit comes from the end hosts. We also reduced the MSS to small sizes and have not observed any difference in the results, MiMBox is always close to plain IP forwarding.

To better understand the results of Figure 4.8, we measure the CPU cycles consumed by each solution. The results are shown in Figure 4.9. To have a fair comparison, we used the 1Gbps links to be limited by the network. Even in this scenario, our solution outperforms the proxies. Figure 4.9 shows the average number of CPU cycles flagged as idle over 10 experiments. It is worth noting that the more cycles are spent in idle mode the more resources are available for other services. The CPU spends more time being idle when using MiMBox. The only exception where MiMBox is never idle is for request sizes of 1KB. In this case, the large number of new connection establishments consumes the CPU cycles as the TCP and MPTCP sockets have to be created. When the request size increases, fewer connection establishments are performed and so fewer CPU cycles are spent for the creation of the sockets. This observation also applies to HAProxy and Squid. However the latter never have any spare idle cycles where the former allows the CPU to have some idle time.

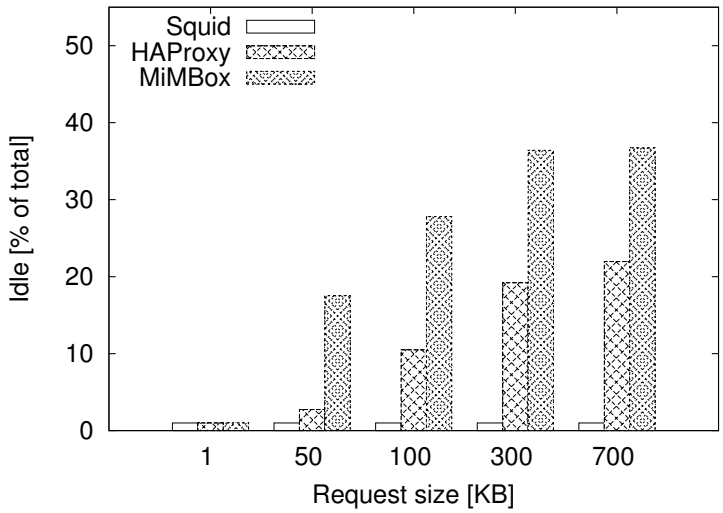


Figure 4.9: MiMBox always uses fewer CPU cycles hence its better performances.

4.5.5 Flow-to-core affinity

We analyze the impact of our flow-to-core affinity extension described in Section 4.4.2 on MiMBox and the application-level `read()`/`write()` converter. We use multiple parallel Netperf sessions to profile the performance with and without flow-to-core affinity. We increase the number of sessions and evaluate the number of cache misses that occur during the transfer. The more cache misses, the worse the performance is as it will cause delay while the CPU stalls until the data is fetched from the main memory. We used Perf [per12] to measure the cache misses, monitoring 20K samples per second.

Figure 4.10 shows the average cache-misses reduction with flow-to-core affinity over 10 measurements. The outcome is twofold. First, a reduction is always introduced when using flow-to-core affinity and the maximum reduction achieved is 60%. Second, the gain of using flow-to-core affinity with the application-level converter is smaller (up to 25 %). This is mainly because most cache misses occur when copying memory from and to the user space. As the application-level converter is running in user space it can be scheduled by the operating system. We measured that the number of context switches is very significant compared to MiMBox. Therefore these context switches can cause cache misses when the user-space application accesses the data.

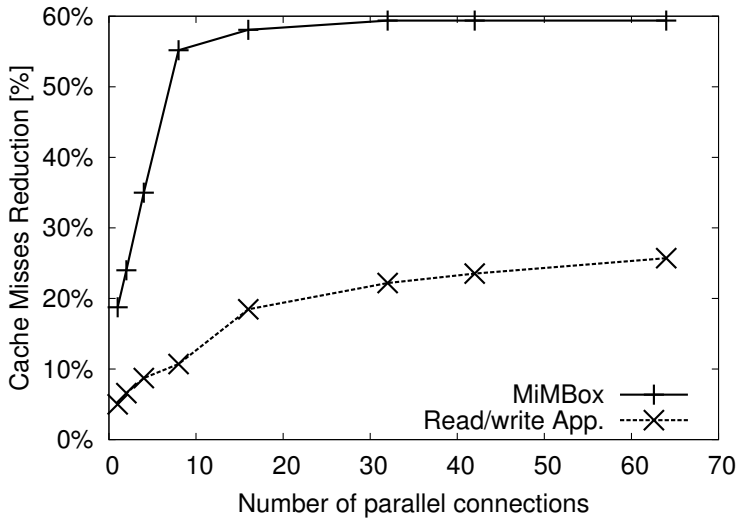


Figure 4.10: The number of cache misses remains constant when activating the flow-to-core affinity while it increases without.

4.5.6 Buffering

By terminating connections MiMBox buffers data. Buffering consumes memory on MiMBox and thus might limit its ability to support a large number of clients. To analyze the impact of the buffer usage we performed two different evaluations where we limit the maximum memory allocated for each socket. For this, we used the system configuration parameters `tcp_rmem` and `tcp_wmem` that allow to configure the minimal, initial and maximum sizes of respectively the send and receive buffers.

We used a real-life simulated environment as presented in Figure 4.6. In this scenario, the client simulates a smartphone that has a WiFi (through ADSL) and a 3G connectivity and the proxy is present in the cloud. The links of the setup used in the previous evaluation where configured so that the client has one link that simulates ADSL with 8Mbps downstream and 512Kbps upstream and 20 msec RTT to the proxy as well as a 3G link with 2Mbps downstream and 256Kbps upstream and 80 msec to the proxy. The link between the proxy and the server is configured with a symmetric 100Mbps and 10 msec RTT with the server. The client side is therefore the bottleneck in this scenario.

We first evaluate the memory consumption of MiMBox when the client transfers at the highest bandwidth available, i.e., 10 Mbps in download and 768 Kbps in upload. For this experiment we used Netperf to measure the bandwidth and varied the `tcp_rmem` and `tcp_wmem`. We use as baseline the mode where there is no proxy between the client and the server.

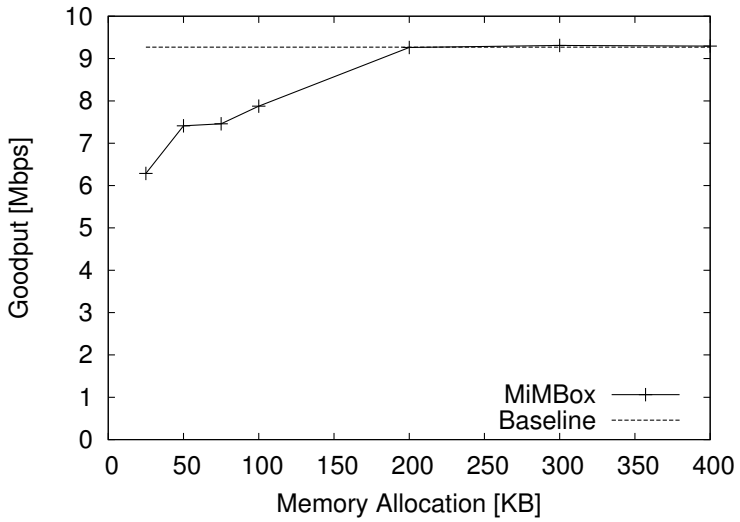


Figure 4.11: Client memory consumption is quite small to achieve a high bandwidth.

Figure 4.11 shows the average download performance over 10 measurements when the memory allocated varies. The client can achieve the highest performance with only 200KB of memory per socket (400 KB in total). This result can be explained by the dimensioning of the receive buffer to $2 \times BDP$ where BDP is the Bandwidth-Delay Product. This ensures application to not stall as it allows the receiver to store out-of-order packets (one BDP) as well as allow the sender to perform fast retransmit. In MPTCP the BDP depends mainly on the slowest subflow:

$$BDP = \sum_{i \in subflows} BW_i \times RTT_{max}$$

In this setup, a receive buffer of $2 \times (8+2)\ Mbps \times 80\ ms = 200KB$ is therefore required to achieve the best performances. A MiMBox having 8GB of memory could therefore scale up to 20,000 connections in this environment[4]. The memory usage could be improved. Indeed, in the download scenario, the socket facing the server does not require 200KB to achieve 10Mbps.

Our second evaluation analyzes the impact of the buffering on interactive applications. For this we used the same environment and added a 1% loss on the client-side to cause retransmissions. We used a custom application [RPB+12] that sends at a fixed 100Kbps rate to simulate an interactive application. We then measured the application delay. Figure 4.12 shows the average delay difference over 10 measurements regarding the minimum delay

[4] 8GB is fairly small. Some Amazon EC2 instances have at most 244 GB of memory. In this case MiMBox could support up to 600k connections.

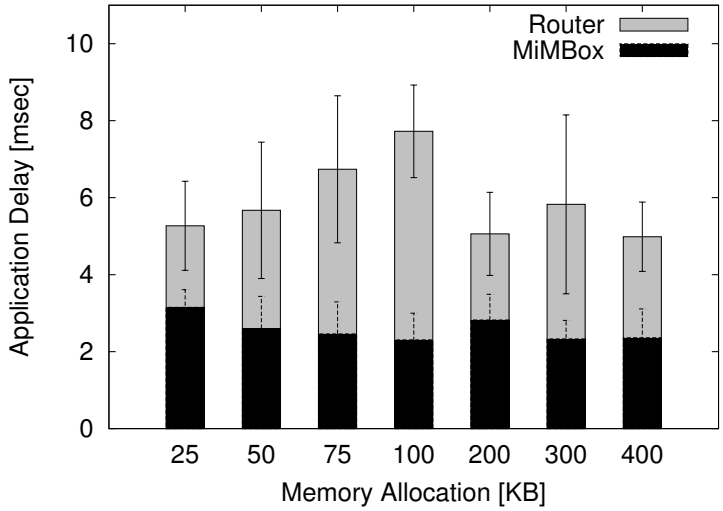


Figure 4.12: On average MiMBox reduces the end-to-end application delay.

(30 msec) observed for various buffer sizes. We can see that MiMBox globally reduces the delay. We observed for a 50KB buffer that there is a 2 msec advantage for the MiMBox. More interestingly the maximum additional delay observed is around 150ms with MiMBox while it is 250ms without. Having a large delay can be problematic for interactive sessions. The reason that MiMBox reduces the delay in this scenario is that by terminating the connection it also allows end hosts to react quickly to losses therefore causing fewer queuing inside the network. A similar tendency has also been observed with other buffer size values as well as when the path between the MiMBox and the server is lossy. We did not observe any significant difference when there is no loss.

4.6 Evolution of MiMBox

As the deployment phase of MPTCP progresses, there will potentially be more clients and servers supporting it. It is obvious that MiMBox does not need to convert the traffic if both end points support MPTCP. In such environments, it is desirable to be able to remove the protocol converter from the data stream when using the explicit mode. Bypassing MiMBox allows to reduce its load and enables a direct communication between the client and the server. In the rest of this section we discuss a fallback mechanism that allows MiMBox to remove itself from the communication. We then evaluate the impact of such fallback.

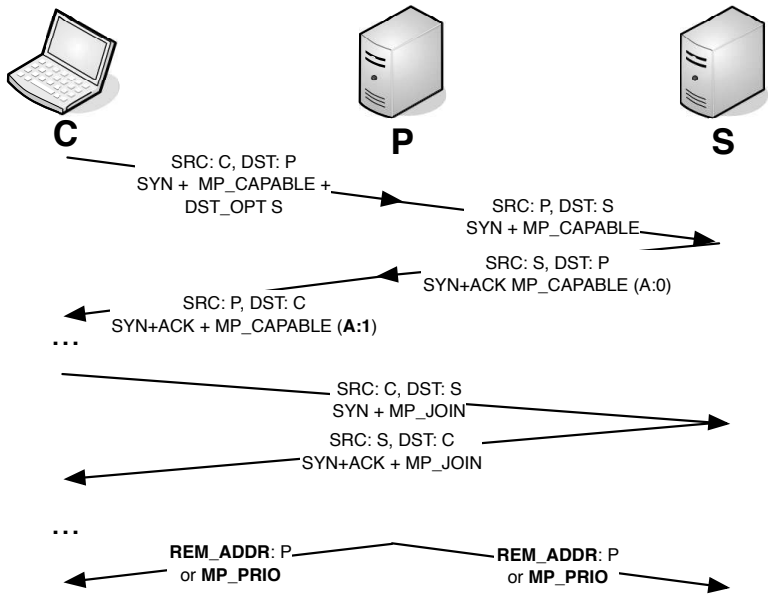


Figure 4.13: MiMBox can remove itself from the communication in order to improve end-to-end delay.

4.6.1 Removing MiMBox from the communication

To remove itself from the communication without disrupting the data stream, MiMBox uses additional mechanisms, presented in Section 1.3, to remove and change the priority of subflows. To remove subflows, the Remove_Addr option is used to signal to the peer that an address has become unavailable. Upon the reception of this option the receiver closes all the TCP subflows that are using this address. MPTCP allows to change the priority of subflows in a binary fashion. By sending the Mp_Prio option or by setting a bit in the Mp_Prio a host can inform the peer that it prefers to not receive data on this subflow (or for this address) if higher priority subflows are available (see Section 1.3.3).

Figure 4.13 shows the operations performed by MiMBox to remove itself from an MPTCP connection. Prior to the removal, there must be at least one established subflow directly between the client to the server.

To shutdown the initial subflow, MiMBox has two choices. Either it completely removes the subflow using the Remove_Addr option or it blocks the peer from sending data on it using the Mp_Prio option. The latter requires to maintain some state to perform the forwarding but gives an advantage if the hosts are mobile. Mobile nodes regularly change their IP addresses. For the stability of the connection it would be useful to always have one *stable* IP address that can be joined by a subflow. MiMBox can

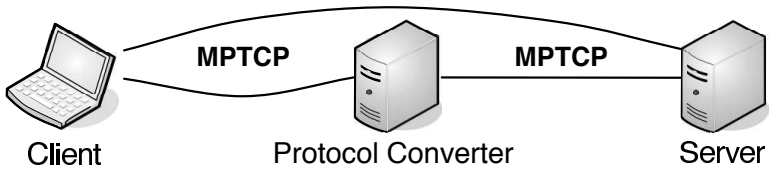


Figure 4.14: The setup used to evaluate the fallback mechanism.

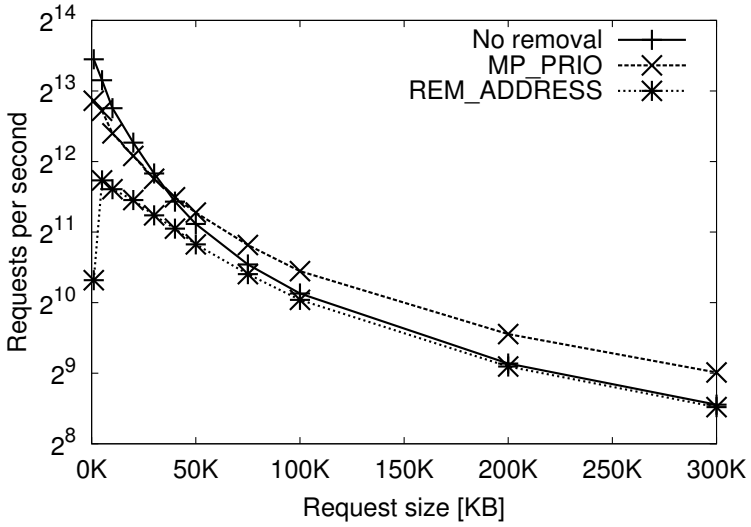


Figure 4.15: Removing MiMBox after a few exchanged bytes results impacts less the performance.

be such a stable point. Both hosts could move at the same time, and still be able to continue the data stream, by creating new subflows passing by the MiMBox. This mobility scenario is however out of the scope of this chapter.

4.6.2 Impact of a MiMBox's removal

To evaluate the impact of the removal of MiMBox from the communication, we implemented the removal as an extension to our implementation that triggers either a MP_PRIO sent within the third ACK of the three-way handshake or a REMOVE_ADDR option when we are sure that MPTCP is working on the master subflow, i.e., when receiving the first data acknowledgement at the MPTCP level. It is not possible to compare our solution with existing ones as existing solutions do not allow such removal. We therefore compare the different solutions to remove the MiMBox. We use a different setup (see Figure 4.14) than in Section 4.5 even though we use the same devices. We take as baseline the MiMBox without removal (and only

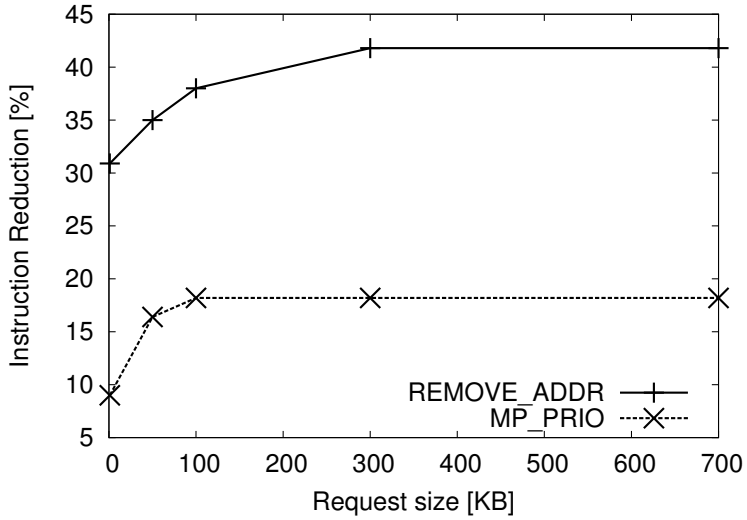


Figure 4.16: The removal introduces a reduction in the number of instructions executed by MiMBox.

one subflow) and compare it with the MP_PRIO and the REMOVE_ADDR solutions. The results of the evaluation are shown in Figure 4.15. The figure shows the average number of requests per second supported (in log-scale) over 10 measurements for different request sizes.

For small file sizes, the removal impacts the performance. In this case, The REMOVE_ADDR has the biggest impact as the server has to potentially reinject already sent data on the initial subflow. For the MP_PRIO solution such reinjection is not needed as the initial subflow still exists and can still acknowledge segments.

For larger file sizes, the impact of the REMOVE_ADDR is negligible. In this case, the limit becomes the network and not the removal mechanism anymore. The time taken to remove itself becomes negligible compared to the time taken to receive the data. The MP_PRIO has an interesting behavior for larger file sizes: it achieves better performance than without removal. This results from a higher congestion on the direct link between the client and the server which delays the establishment of the second subflow. This delay allows to send more data on the initial subflow before the establishment of the second one.

For MiMBox, it is therefore worth to wait until a few kilo-bytes have been exchanged before starting the removal procedure. It is even more important for the REMOVE_ADDR mechanism. Removing the MiMBox from a flow is thus more interesting for long-lived flows.

Another advantage of removing MiMBox from the communication is

that it reduces its CPU utilization, therefore enabling it to handle more connections. Figure 4.16 shows the average reduction in the number of instructions over 10 measurements executed by the MiMBox using the two different removal modes. For small file sizes, the reduction is small as the second path is almost never used. For larger sizes, the reduction increases up to a maximum, 18% for the MP_PRIO and 40% for REMOVE_ADDR. This maximum results from the fact that for large file sizes a MiMBox needs to handle fewer connection establishments. The two modes have different performance improvements: the MP_PRIO still uses the initial subflow for some time before switching to the second one, therefore causing more cycles to be used.

4.7 Related work

The end-to-end principle assumes that TCP connections and all application-specific functions ought to reside in the end hosts. This is less and less true as the Internet is dominated by middleboxes [HNR+11]. Starting in the 1980s, researchers have proposed protocol converters that terminate one transport connection on one side and initiate another one on the other side [Gro06]. This idea has been applied by various authors for various reasons. A first use case are wireless networks where end-to-end performance can be improved by splitting the TCP connection [BB95] and using improved retransmission techniques on the wireless link [BPSK97, BKG+01]. Our protocol converter allows wireless hosts to use different wireless interfaces at the same time even when the servers do not support MPTCP.

Another example are the HTTP proxies that also terminate TCP connections. These proxies are either deployed close to the end user [hap12, squ12] or in the cloud to improve access to data centers [PWH+10, Apa13]. Various techniques have been proposed to improve their performance [RR02, SHHP00]. In a nutshell, these solutions forward data in the application at the beginning of the connection and can instruct the kernel to autonomously forward packets afterwards. HAProxy [hap12] is an example of such a modern high-performance proxy. Our protocol converter is different. First, its is entirely implemented in kernel space, which improves performance for both short and long flows as demonstrated by our experimental evaluation. Second, it converts MPTCP to regular TCP and vice versa. This implies that our protocol converter handles the specificities of MPTCP like segment reordering, checksums, etc. This is more complex than passing regular TCP segments in both directions. Finally, our converter is also agnostic of the application used.

Recently, a few authors have proposed some kind of MPTCP proxies.

Raiciu et al. [RNBH11] discuss the role that such proxies could play to support mobile hosts. Ayar et al. [ARBW12b] propose to use multiple paths inside the core network to improve performance while still using regular TCP between the end hosts with their proposed Splitter/Combiners. Hampel and Klein [HK12] propose MPTCP proxies and anchors as well as some extensions to the MPTCP protocol to support them. A prototype implementation of such proxy exists with an incomplete support of MPTCP [AL13], i.e., only one subflow at a time is supported for the client.

4.8 Conclusion

Deploying a new TCP extension like MPTCP can be difficult despite its clear benefits for users. We propose MiMBox to allow clients to already benefit from MPTCP during its deployment phase. MiMBox efficiently translates MPTCP to TCP. By implementing it entirely in the Linux kernel, we achieve high performance on commodity x86 servers. Furthermore, we show how MPTCP can be leveraged to allow MiMBox to remove itself from the communication when both communicating hosts are MPTCP-capable.

We compare the performance of MiMBox with existing HTTP proxies and simpler designs, e.g., using TCP Splice. Our evaluation shows that MiMBox overcomes existing proxies both when handling long TCP flows and when serving a large number of HTTP clients. From a performance viewpoint, MiMBox is close to the performance of an IP router. Furthermore, MiMBox uses fewer CPU cycles than other solutions while providing better performance. Moreover, it uses flow-to-core affinity to steer packets from the TCP and MPTCP connections to the same core which leads to fewer cache misses and thus increases the overall performance.

Chapter 5

Revisiting flow-based load balancing

5.1 Introduction

Load balancing plays a key role in enterprise, data center and ISP networks. It improves the performance and the scalability of the Internet by distributing the load across network links, servers, or other resources. Load balancing allows to maximize the throughput [Hop00], achieve redundant connectivity [ICBD04] or reduce congestion [CWZ00]. Different forms of load balancing are deployed at various layers of the protocol stack. At the datalink layer, frames can be distributed over parallel links between two devices [Ass08]. At the application layer, requests can be spread on a pool of servers [KKSB07].

At the network layer, the most common technique, *Equal Cost Multi-Path* (ECMP) [Hop00, CS13d], allows routers to forward packets over multiple equally-good paths. ECMP may both increase the network capacity and improve the reaction of the control plane to failures [ICBD04]. Current ECMP-enabled routers proportionally balance flows across a set of equal-cost next hops on the path to their destinations. The most deployed next-hop selection method is solely based on a hash computed over several fields of the regular packet headers [Cis98, Hop00]. Using a hash function ensures a somewhat fair distribution of the next-hop selection [CWZ00]. Most implementations include the transport-level port numbers to improve the load spreading while preserving the packet sequence of transport-level flows [CWZ00].

Using the transport header to influence the network-layer forwarding breaks the layered architecture. Although some topology discovery appli-

cations exploit this principle [ACO+06, AFT07], it remains difficult for end hosts to accurately control forwarding decisions or even be aware of the path diversity that is available in the network. In Section 1.3, we saw that subflows from MPTCP can have different 5-tuples and so might follow different paths in the network. There is therefore a benefit for multipath-aware transport protocols as Multipath TCP (and even for SCTP-CMT [IAS06]) to be able to select network parts that are as independent as possible to avoid congestion [BAAZ10]. Unfortunately, this is not possible with existing hash-based load-balancing techniques.

Data center designs rely heavily on ECMP [AFLV08, GHJ+09, GLL+09, MYAFM10] to spread the load among multiple paths and reduce congestion. This form of load balancing is blind and naive. Congestion can still occur inside the data center and lead to reduced performance. Data center traffic is composed of mice and elephant flows [BAM10, KSG+09]. Mice flows are short and numerous but they usually do not cause congestion. Most of the data is carried by a low fraction elephant flows. Based on this observation, several authors have proposed of traffic engineering techniques that allow to route elephant flows on non-congested paths (see [BAAZ10, CKY11, AFRR+10] among others). Those techniques often rely on OpenFlow switches [MAB+08] to control the server-server paths. Unfortunately, the scalability of such approaches is limited, which may lead to an overload of the flow tables on the OpenFlow switches [CKY11, CMT+11].

In this chapter, we show that another design is possible to exploit the path diversity that exists in data center networks without maintaining any state within the network. Current hash-based implementations rely on the IP and TCP headers to select the load-balanced path over which each flow is forwarded. It is therefore very difficult for a host to predict the path that a flow will follow. We show in this chapter that hash functions are not the only way to practically enable path diversity. We propose a new deterministic scheme called *Controllable per-Flow Load Balancing* (CFLB) that allows hosts to explicitly select the load-balanced path they want to use for a specific flow. CFLB allows multipath aware end hosts to deterministically select the path followed by their packets inside the network, without any change to the IP and TCP headers.

CFLB is designed such that it meets the following requirements: *(i)* all packets of the same flow follow the same path; *(ii)* no overhead; *(iii)* transparent to non-CFLB aware sources; *(iv)* operate at line rate. Thereby, CFLB enables sources to encode, using an invertible function, inside existing fields of the packet header the load-balanced path that each packet should follow. Each router uses the same invertible function to decode the forwarding decision it needs to apply to the packet. This simple mechanism can be implemented efficiently as to enable line rate forwarding. CFLB is

transparent for applications that do not want to steer packets and does not require any change for non-CFLB-aware end hosts. CFLB does not require any extension in packet headers and does not store any state in routers which perform only simple calculations. Instead, the state is stored in end hosts and conveyed using header fields in each steered packet.

To summarize, our key contributions are:

- A study of path diversity in data center and ISP networks;
- The design of a packet steering solution that is stateless for routers and does not require a shim header;
- The evaluation of the benefits of combining such a solution with Multipath TCP.

The remainder of this chapter is organized as follows. We first recall in Section 5.2 current hash-based load-balancing basics. Section 5.3 discusses related work. We then provide a detailed description of the operation of CFLB in Section 5.4. In Section 5.5, we analyze the performance of CFLB. We first use trace-driven simulations to compare CFLB with existing hash-based techniques and, then, we implement CFLB in the Linux kernel to evaluate its packet forwarding performance. We also evaluate the benefits of CFLB for Multipath TCP hosts. In Section 5.6, we discuss other possible applications of CFLB.

5.2 Network-based load balancing in a nutshell

In this section, we first provide an in depth overview of the network-layer load balancing using hash-based techniques as well as an evaluation of path diversity in data centers and ISP networks. Finally, we discuss the interactions between MPTCP and load balancing.

5.2.1 Path diversity at the network layer

There exist several proposals to enable path diversity at the network layer [MFB+11]. In practice *Equal Cost Multi-Path* (ECMP) [Hop00] is currently the mostly deployed one. ECMP is both a path selection scheme and a load distribution mechanism. To enable path diversity, it uses paths that tie to ensure loop-free forwarding. According to the level of resulting path diversity, routers then proportionally balance packets over their multiple next hops. This proportional aspect is an arbitrary design choice which is not in the scope of this chapter. We focus on the practical implementation

of the mapping (packet → next-hop). Various next-hop mapping methods exist to practically balance packets over load-balanced paths.

An ECMP load balancer should fairly share the load over the next hops. Since flows vary widely in terms of number of packets and volume, this is not that easy [Hop00, CWZ00]. Most load-balancing methods that meet these requirements are based on a hash function [CWZ00, Cis98]. They compute a hash over the fields that identify the flow in the packet headers. These fields are usually the source and destination IP addresses, the protocol number and the source and destination ports, i.e., the *5-tuple*. The computed hash can then be used in various ways to select a next hop. The simplest and most deployed method is called *Modulo-N*.

If there are N available next hops, the remainder of dividing the hash by N is used as an identifier of the next hop to use. The following equation is performed to compute the remainder L (where $\mathcal{H}$ corresponds to the hash function):

$$L = \mathcal{H}(IP_{src}, IP_{dst}, Prot, Port_{src}, Port_{dst}) \; mod \; N$$

L is then used to select a particular outgoing path, generally the L^{th} path among the N paths set is chosen.

Note that this naive hash-based technique is, however, subject to a problem called *polarization* [CS05, MMH07]. This problem comes from the fact that if each router applies the same hash function on the same input, they will take the same decision. Polarization degrades therefore the efficiency of the load balancing. A simple solution is to use a different hash function in each router or to add a value in the input of the hash function, usually a unique *router id*.

There is no standard defining the hash-based function that should be used for this computation and different hash functions are implemented by router vendors. CRC or Internet checksums [CWZ00] are typical ones as they are already implemented in hardware on all routers. Cao et al. [CWZ00] show that using the *16-bit CRC* (CRC-16) algorithm on the 5-tuple gives the best results in terms of load balancing distribution. Hopps [Hop00] show that hash-based ECMP provides the best cost/performance tradeoff.

Due to the non-determinism property of hash functions, forcing a path in a load-balanced network is hard. In general hosts can only vary the transport header fields to try to influence the path selection [ACO+06]. We give more detail in Section 5.2.3.

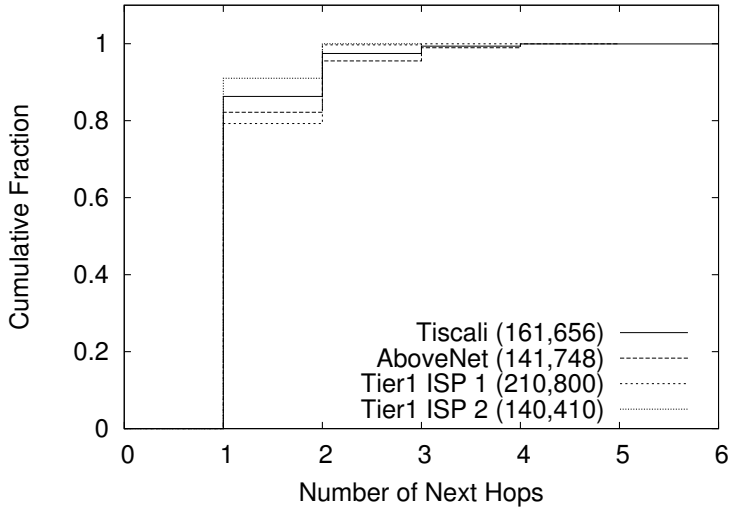


Figure 5.1: Local next hop diversity in ISP networks.

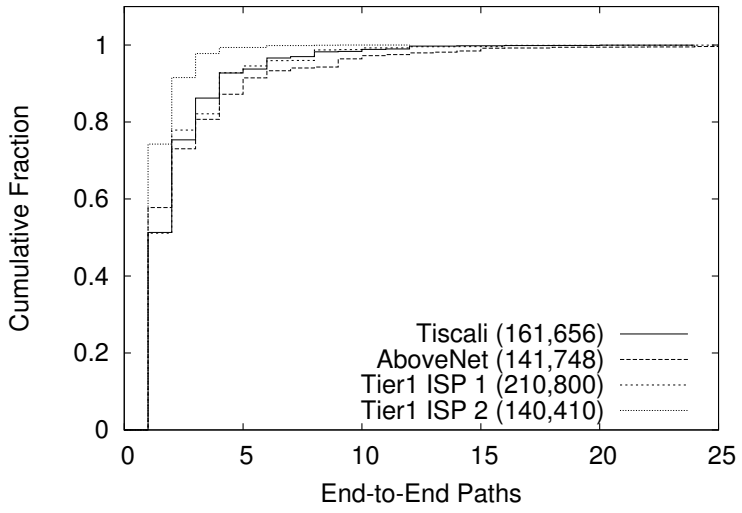


Figure 5.2: Global path diversity in ISP networks.

5.2.2 Path diversity in ISPs and data center networks

Experimental measurements by Augustin et al., reported in [AFT07], show a wide deployment of multipath routing strategies in the Internet. From measurements between 15 sources and over 68,000 destinations, they found that 39% of these source-destination pairs have at least two load-balanced paths. The utilization of load balancing in ISP networks has also been studied in [MFB+11].

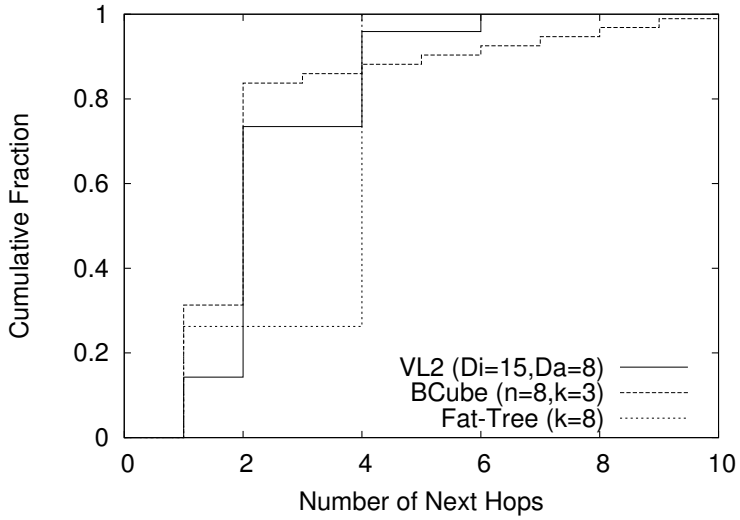


Figure 5.3: More than 80% of the server pairs in popular models of data center topologies have two or more paths between them.

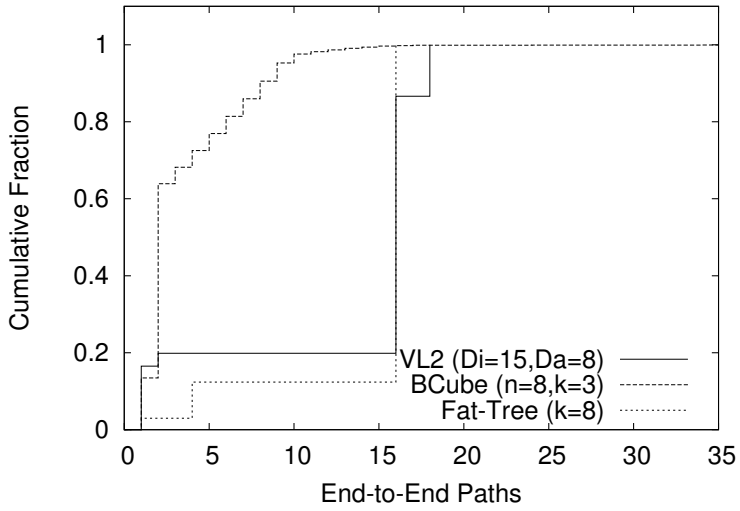


Figure 5.4: On average, for 70% of destinations, routers and switches have multiple next hops.

ECMP is widely used by IP and data center networks either for load balancing or failure recovery purposes [ICBD04]. To demonstrate the potential presence of ECMP load balancers, we analyze the topology and the resulting routing of several ISP and data center networks. In particular, we focus on the number of routers having equal-cost paths. For ISP networks, we use two kinds of topologies: *(i)* Tiscali and AboveNet come from the

Rocketfuel topologies database [SMWA04] on which IGP weights have been inferred, *(ii)* ISP 1 and 2 are real Tier1 ISP topologies anonymized for confidentiality purposes. Their sizes in terms of nodes and edges are given in the figure captions. For data center networks, we use three generic models, Fat-Tree [AFLV08], VL2 [GHJ+09] and BCube [GLL+09]. For each model, we pick one representative topology giving the same amount of servers, approximatively 600 servers per topology. The model parameters are given in the caption (the parameters are the same as defined in [AFLV08], [GHJ+09] and BCube [GLL+09]). In BCube's design, servers do not only act as end hosts, they also act as relay nodes for each other, we therefore consider servers to evaluate the local next hop diversity. Since the simple spanning tree at the link layer also tends to be replaced by multipath-capable routing [TP09], we also consider switches as load balancers. Furthermore, for data center networks in general, we only consider server as destinations.

Figures 5.1 and 5.3 show the local next hop diversity, i.e., the number of next hops per router-destination pair, of ISP and data center networks. Figures 5.2 and 5.4 provide a global path diversity analysis, i.e., the number of end-to-end path between each source and destination pair, for the same networks. The results obtained on these two families of networks differ because of their architectural design. Data centers are really structured networks favoring the presence of equal cost paths whereas ISP networks do not present such regular characteristics. In Figure 5.1, we notice that fewer than 20% for ISP1 and fewer than 10% for ISP2 of routers-destinations pairs enable ECMP multipath local diversity, with an upper-bound of 6 forwarding next hops for a couple router-destination in AboveNet. However, considering an end-to-end perspective, Figure 5.2 shows that between 25% and 50%, respectively for Tiscali and ISP2, of the source-destination pairs can benefit from at least one ECMP capable router (with a pair source-destination having a maximum of 76 available paths for Tiscali, Figures 5.2 and 5.4 x-axis are truncated for clarity reasons). This difference results from the fact that the probability of benefiting from at least one ECMP router exponentially increases with the path length. In such ISP networks, the distribution of path length follows a normal distribution with a mean of half the diameter (with a maximum diameter of 16 hops in Tiscali).

For the data center networks, the presence of equal cost paths is much more important (see Figures 5.3 and 5.4) than in ISP networks. On average, switches are able to perform local load balancing among a set of two or more next hops for 70% of destinations, while more than 80% of the pairs of servers pass through at least one ECMP load balancer (with a maximum of 64 available paths for the BCube topology). In practice, paths in DC are really short among servers since this kind of topologies have small diameters and are locally meshed.

5.2.3 Interactions with Multipath TCP

In MPTCP, subflows can be established between the same IP addresses with different ports or between different addresses of the same end hosts [FRHB13]. Since subflows have different 5-tuples, each can be load balanced over a different path. Raiciu et al. evaluated Multipath TCP inside data centers in [RBP+11]. Simulations and measurements show that performance improves when Multipath TCP is allowed to use multiple subflows in such data centers. Measurements collected in data centers [BAM10] show that the traffic load can change quickly and performance could be improved by dynamically routing traffic around congested links [BAAZ10]. To achieve better performance, MPTCP would need to open subflows that use less congested links. A possible solution to achieve this goal would be to rely on a network information server similar to the ALTO solution being developed within the IETF [APY13]. This server could collect network statistics and then inform MPTCP of the best load-balanced paths to be used to reach a given destination. However, MPTCP establishes subflows on random port numbers. With standard load balancing, there is no guarantee that a different path will be chosen for each of these subflows.

As the 5-tuple of each TCP subflow is randomly generated, there is a chance for sources that the subflows do not take the non-congested load-balanced paths. Indeed, let us consider a source has m load-balanced paths towards a destination and that l of those m paths are non-congested. If all paths are equiprobable, then the probability that k subflows go through k different paths amongst the l non-congested paths is defined by:

$$P^{m}_{(k,l)} = \frac{l!}{(l-k)! \times m^k} \qquad (\forall\ k,l,m \in \mathbb{N} \mid k \leq l \leq m) \tag{5.1}$$

If there are 16 load-balanced paths, which seems realistic with respect to the observations we made in the previous section, the probability to cover 4 non-congested paths is low, e.g., if a source generates either 2 or 4 subflows the probability is respectively $P^{16}_{(2,4)} = 6.7\%$ and $P^{16}_{(4,4)} = 0.5\%$. In practice, the equiprobability assumption between load-balanced paths may not hold such that actual figures may be worse. The load-balanced paths may be unbalanced such that non-congested paths may be more difficult to setup using a random approach.

One could argue that sources could generate as many subflows as possible to try to increase the probability displayed in Equation 5.1. However, establishing additional subflows comes with a cost, as each new subflow requires a three-way handshake with crypto-authentication before being established and each additional subflow increases the requirements in memory at the receiver due to a larger receive-buffer [BPB11]. From a performance

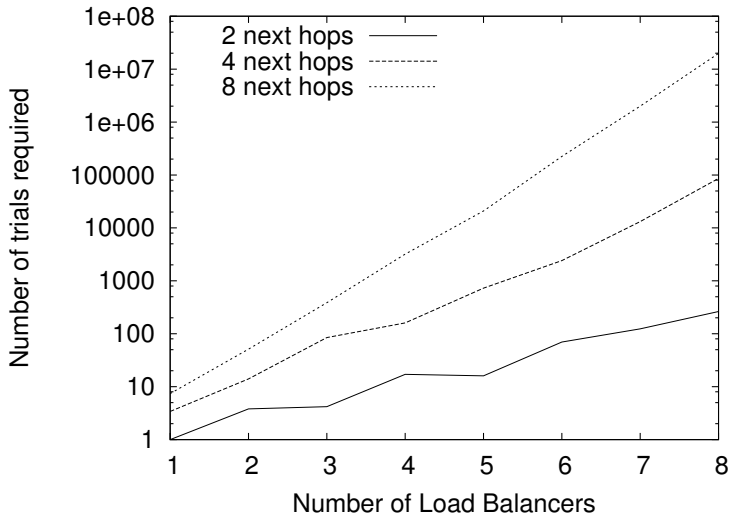


Figure 5.5: A brute force approach is only possible if we know the hash function used by the routers. It might also require a large number of CPU cycles if the network has a large number of load balancers.

viewpoint, Multipath TCP would obviously benefit from being able to establish the minimum number of subflows to efficiently utilize the less congested load-balanced paths. This, however, requires the ability to map deterministically a subflow to a path in order to setup subflows on non-congested paths.

A brute force approach could also be used to find the right port to use to force a subflow to follow a path. This however requires to know the hash function in routers, to the best of our knowledge, we did not find the hash function used by major vendors. Nevertheless, if we know the function in used in the routers, we looked at how many trials are required before finding the right combination of ports with a random approach. Figure 5.5 shows that if the network contains a large number of load balancers the number of trials can quickly explode and consumes a large amount of CPU cycles for each subflow.

5.3 Related work

Several researchers have evaluated the performance of dynamic load-balancing techniques. Menth et al. consider in [MMH07] dynamic load balancing at the network layer not solely based on static forwarding ratio [CWZ00]. Kandula et al. propose another dynamic load-balancing technique in [KKSB07].

Network-layer load balancing is not limited to the use of ECMP. Other forms of multipath routing have also been proposed. BANANAS [KKW+03], Routing Deflections [YW06], Path Splicing [MEFV08], and Pathlet Routing [GGSS09] are examples of scalable routing primitives that allow end systems to use non-shortest paths as an alternative to explicit source routes. These solutions rely on the utilization of a shim header to allow end hosts to exploit the path diversity. Otherwise, the packet is forwarded along the normal route.

Xi et al. proposed in [XLC11] *Probe and RerOute Based on ECMP* (PROBE). This technique combines path probing, similar to Paris-traceroute [AFT07], to discover the ECMP paths and uses a NAT technique on the hosts to reroute the flows.

Recent proposals have focused on ways to distribute the flows inside data centers [CKY11, AFRR+10, MYAFM10]. *SPAIN* [MYAFM10] is a proposal that configures multiple static VLANs in data centers to expose to end hosts the underlying network paths. In the case of *SPAIN* either the hosts or the switches must be modified to allow selecting a path for each flow.

Other solutions have looked at how to dynamically allocate flows to paths, Al-Fares et al., among others, show that it is possible to avoid generating such hot-spots by efficiently utilize aggregate network resources by using *Hedera*, a dynamic flow scheduling system [AFRR+10]. The idea behind this solution is to assign large flows to lightly loaded paths by querying a scheduler that possesses informations about the current network load. Hedera stores state in the switches, i.e., flow entries in OpenFlow [MAB+08], in order to ensure that flows follows the desired path. However, as pointed out by Curtis et al. in [CKY11], there currently does not exist any OpenFlow switch that can support the required number of flows at each rack switch.

5.4 Controllable per-Flow Load Balancing

CFLB enables hosts to select a load-balanced path among the diversity offered by the network layer. Compared to a source routing solution, it allows CFLB-aware sources to steer their packets inside the network without requiring any header extension. Figure 5.6 presents the overview of the mechanisms ECMP and CFLB use to select a next hop when several are possible. The mechanisms are quite similar: for IPv4 networks both apply an operation on the 5-tuple to select a next hop. ECMP uses a hash-modulo operation while CFLB uses a more complex operation that will be detailed in the remainder of this section. CFLB uses an additional field of the packet

header, the *Time to Live* (TTL), as a way to "identify routers". CFLB is not only limited to layer-3 routing, it can work in any environment where a hop count is used (e.g., TRILL [TP09], MPLS, IPv6, etc.). CFLB decomposes those header fields in two categories: *controllable* and *uncontrollable*. The controllable fields are the fields that can be selected by the source, e.g., the source and destination TCP ports [1] while the remaining fields are uncontrollable, i.e., the protocol number and the source and destination addresses. In short, the controllable fields are used to convey the path selector while the uncontrollable fields are used to add randomness for non-CFLB aware flows. The CFLB mechanism can be decomposed in four separate operations:

1. The desired path is specified by the source as a sequence of next-hop selections, that we call a *path selector*;
2. This path selector is then encoded by the source inside the *controllable* header fields;
3. Each router extracts from these controllable fields the encoded path selector;
4. The path selector is then used to extract the next-hop selection the routers need to apply to load-balance each packet.

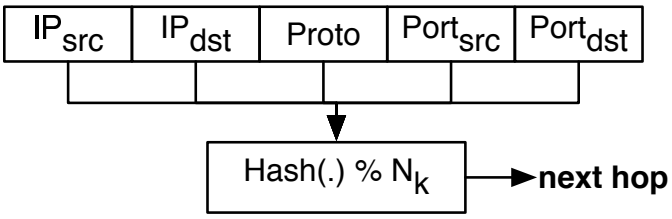
This section is organized to help understand the design choices behind CFLB. We first describe the path selector. We then describe operations (1) and (4) and finally operations (2) and (3).

5.4.1 Path selector

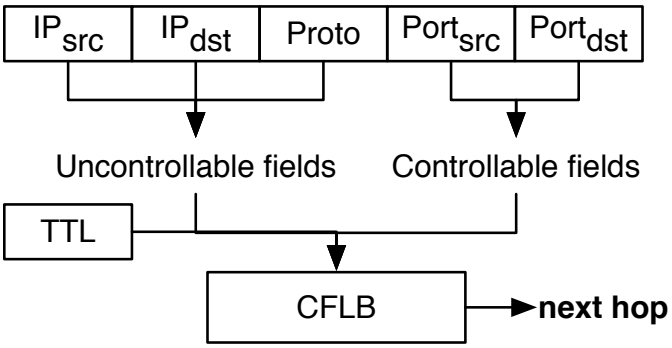
In a network offering path diversity, there exist multiple paths having the same cost between a source and a destination. We call these load-balanced paths. Figure 5.7 shows all load-balanced paths between a source S and a destination D in a simple network. As only load-balanced paths are shown, Figure 5.7 does not show links that are not used to route packets to the destination D. In this example there exist five different load-balanced paths between nodes S and D. Table 5.1 lists the notations used in this chapter and their definitions.

CFLB allows sources to control the next-hop selection in routers when they have more than one possible next hop for a destination. For example, router R_4 has 3 potential next hops for destination D. By knowing the number N_k of available next hops towards a destination for each router k in the network, next-hop selections can be mapped to a number $n_i \in [0, N_k[$,

[1] In MPTCP additional subflows can use a different destination port than the initial subflow as they are identified by the token present in the MP_JOIN option.



(a) Regular ECMP.



(b) CFLB.

Figure 5.6: Overview of load-balancing mechanisms.

Symbol	Definition
B	The radix of the path selector, i.e., the numeral base to encode the path selector.
n_i	The next-hop selection of the i^{th} positioned router.
L	The length of the path selector, i.e., number of next-hop selections that can be encoded in it.
N_k	The number of load-balanced next hops available at a router k (for the sake of clarity, we ignore the destination prefix).
$F(x)$	The invertible function applied on x.
$H(x)$	The Hash function applied on x.
c_f	The controllable fields used.
u_f	The uncontrollable fields used.
$E_i(n_i)$	The function applied on a next-hop selection, it adds "randomness" using u_f.
$D_i(x)$	The function that performs the inverse of E_i, i.e., $D_i(E_i(x)) = x, \ \forall x \in [0, B[$.
$A\|\|B$	The concatenation of A and B.

Table 5.1: General notations.

where n indicates the index of the next hop which should be selected. Using such mapping, if source S wants to follow the highlighted path in Figure 5.7, the next-hop selection for router R_4 is 2. Note that only routers having multiple load-balanced next hops to forward a packet must be controlled by

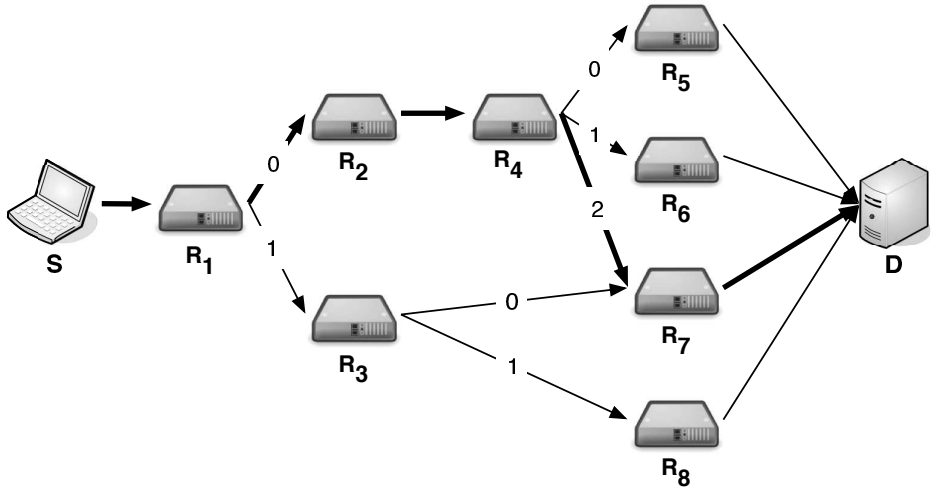


Figure 5.7: The load-balanced paths between S and D.

the source if the latter wants to follow a specific path in the network.

To specify the next-hop selection a router needs to perform, it must be identifiable by the source. As we only want to rely on existing fields of the packet header, we use the TTL field to identify the position of a router inside the path selector. The TTL is set by the source and decremented by each router that forwards the packet. The field can thus be used by a router to identify its relative position compared to the initial TTL value and thus to represent its position inside the path selector. Therefore, we define the position of each router within the path selector to be $i = ttl \mod L$ where ttl is the TTL value of the packet received at the router and L is the number of next-hop selections that can be encoded in the path selector. The source can, by knowing the initial TTL of the packets[2], encode the next-hop selections of each CFLB router on the path at their corresponding position inside the path selector.

Let us assume that source S uses an initial TTL of 64, that the length of the path selector is $L = 20$ and that the source wants its packets to follow the highlighted path in Figure 5.7. The positions of the next-hop selections of routers R_1 and R_4 inside the path selector are respectively 4 ($64 \mod 20$) and 2 ($62 \mod 20$). The complete load-balanced path can therefore be expressed as the following sequence of next-hop selections: $\{(4 \rightarrow 0), (2 \rightarrow 2)\}$ where the notation $(x \rightarrow y)$ refers to a router at position x which should select its y^{th} next hop.

[2]Most operating systems use a system wide default TTL.

5.4.2 Representation and extraction

CFLB stores a path selector as a positional base-B unsigned integer, where B is known as the radix and shared by all the nodes in the network. This maximizes the number of next-hop selections that can be encoded inside the path selector, while minimizing the number of bits used. As the potential value of n_i is bounded by the radix B, one should choose a B value so that it is the maximum of the number of available next hops N_k towards each destination in every routers of the network. In practice, most routers have a hardcoded upper bound on the number of next hops they can use for a destination, most of them support up to of 16 next hops [AFT07].

Representation

A path selector p can be generalized as:

$$p = \sum_{i=0}^{L-1} n_i \times B^i \tag{5.2}$$

Where n_i is an unsigned integer in base B that represents the next-hop selection of the router having the i^{th} position within the path selector. If a source wants its packets to follow the same path for different services then it might cause a path selector collision. To overcome this issue, CFLB generates multiple path selectors to describe the same load-balanced path using two solutions that can be combined. First, the unused positions inside the path selector can be filled with random values (as for R_2 and R_7 in Figure 5.7). Second, by changing the initial TTL, the position of each router in the path selector changes and thus the path selector value.

Extraction

A router receiving a path selector p can extract the forwarding decision n_i by first extracting the *ttl* of the packet and then computing $i = ttl \mod L$. Finally, n_i can be extracted by applying Eq. 5.3.

$$n_i = \left\lfloor \frac{p}{B^i} \right\rfloor \mod B \tag{5.3}$$

The integer division by B^i removes all load-balanced next-hop selections of upstream routers while the modulo operation removes all load-balanced next-hop selections of downstream routers.

Length

The fixed size of the packet header fields used to encode the path selector limits the number of encodable next-hop selections to:

$$L = \lfloor \log_B(2^X) \rfloor \tag{5.4}$$

where X is the size in bits of the header fields used to encode the path selector.

Example

Let us now illustrate how CFLB works in the simple network shown in Figure 5.7. Assume that the source uses an initial TTL of 64 and wants this packet to follow the bold path in Figure 5.7. Furthermore, B is 3 and L is 20. The path selector computed by the source based on Eq. 5.2 can therefore be expressed as:

$$p = \{(4 \rightarrow 0), (2 \rightarrow 2)\} = 0 \times 3^4 + 2 \times 3^2 = 18$$

Note that this implies encoding a next-hop selection of 0 for all other positions inside the path selector. This value is then encoded inside the packet header (we discuss how the path selector is encoded and decoded in the header fields in Section 5.4.3). R_1 retrieves from the packet header the same path selector and the TTL to compute its position inside the path selector, i.e., 4. It then computes the next-hop selection it needs to apply on the packet by using Eq. 5.3:

$$n_4 = \left\lfloor \frac{18}{3^4} \right\rfloor \mod 3 = 0$$

R_1 decrements the TTL of the packet and forwards it to the next hop labeled 0, i.e., R_2. R_2 does not have load-balanced next hops. It forwards the packet to R_4 and decrements the TTL. R_4 applies the same operation as R_1. R_4 computes:

$$n_2 = \left\lfloor \frac{18}{3^2} \right\rfloor \mod 3 = 2$$

The packet is therefore forwarded to R_7 and then to D as R_7 only has one next hop to forward the packet.

5.4.3 Encoding and decoding the path selector

CFLB uses the controllable fields of the packet header in order to convey the path selector from router to router. These fields must not change

during the forwarding process. We use an invertible function F to encode the path selector. Indeed, a simple bijective function might end up in a poor distribution of the non-controlled traffic [CWZ00]. For instance, if only the port numbers are used as controllable fields, we do not want all web traffic to go through the same next hop. Therefore, we require that the invertible function exhibits the *avalanche effect* [WT86], that is for a small variation of the input (different source-ports) a large variation of the output is observed. Based on this requirement, block ciphers such as Skip32[3] or RC5 [Riv95] are good candidates to implement this invertible function when 32 bits are controllable. When more bits are available, we recommend to use a block cipher mode of operation as *Format-Preserving Encryption* (FPE) construction, such as the *eXtended CodeBook* (XCB) mode of operation [MF04] that accepts arbitrarily-sized blocks, provided they are as large as the blocks of the underlying block cipher. Depending on the number of bits that are controllable in the packet header, different types of block ciphers can be used with XCB, from 32-bit symmetric-key block cipher to most common ones such as DES, 3DES or AES. Furthermore, efficient hardware-based implementations of such block ciphers exist [CDK09, HV04]. Using such functions to encode the path selector enables a router to apply the inverse of this function on the controllable fields of the packet header to retrieve the path selector.

Uncontrollable fields

CFLB is designed based on the properties of hash-based load balancers. It must be transparent to sources that do not need to control the load-balanced path taken by their packets. In this case, the controllable fields are random and do not encode a path selector. CFLB must still distribute such packets efficiently among the available load-balanced next hops. As Cao et al. showed in [CWZ00], the most efficient packet distribution is achieved when all the headers fields identifying a flow are used as input to the load-balancing function. CFLB therefore uses also the uncontrollable fields, source and destination addresses and the protocol number, as input. For that, the way the path selector is encoded slightly changes from Eq. 5.2:

$$E_i(n_i) = (n_i + H(u_f)) \mod B \tag{5.5}$$

$$p = \sum_{i=0}^{L-1} E_i(n_i) \times B^i \tag{5.6}$$

Where H is a hash function and u_f contains the uncontrollable fields of the packet header. This allows to efficiently distribute packets over available

[3] `http://www.qualcomm.com.au/PublicationsDocs/skip32.c`.

next hops of each router, while still allowing routers to recover the next-hop selections encoded by the sources.

Eq. 5.6 can be inverted to find the next-hop selection n_i to apply on the router whose position is i by applying the following operation on the path selector extracted from the packet header:

$$D_i(x) = (x - H(u_f)) \mod B \tag{5.7}$$

$$n_i = D_i(\left\lfloor \frac{p}{B^i} \right\rfloor) \tag{5.8}$$

Next-hop selection

The next-hop selection n_i is a value comprised between 0 and $B-1$. To select a next hop, CFLB applies a mapping between n_i and a value between 0 and $N_k - 1$. However, as $N_k \leq B$, an issue arises when using a simple modulus operation when B is not a multiple of N_k. In this case, the load balancing distribution might be poor. For instance, if $N_k = 2$, $B = 3$, and the input is uniformly distributed, then the router ends up forwarding 75% of the incoming packets to the first next hop. To resolve this problem, CFLB computes the next-hop selection n_i for the packet as follows :

$$n_i = \begin{cases} D_i(\left\lfloor \frac{p}{B^i} \right\rfloor), & \text{if } D_i(\left\lfloor \frac{p}{B^i} \right\rfloor) < N_k \\ H(c_f \| u_f) \mod N_k, & \text{otherwise.} \end{cases} \tag{5.9}$$

The intuition behind Eq. 5.9 is that CFLB must distinguish whether the packet was controlled by a source or not. If the packet was indeed controlled, the next-hop selected, n_i, must be the one encoded in the path selector. However, the non-controlled packets must be distributed randomly among the N_k available next hops. In Eq. 5.9, if the packet to forward is a controlled one, then the resulting next hop to select should be lower than N_k (the number of next hops in the routing table for the packet's destination), the decision encoded at the source is correctly taken. Otherwise, it means that the packet is not a controlled one, resulting in a random distribution of the packet on one of the N_k available next hops.

Topological changes

In case of topological changes (transient or permanent), a CFLB-router will renumber indexes, i.e., update the $n_i \rightarrow R_j$ mapping and N_k, of its current available next hops towards each destination. In such a case, while

new flows are "aware" of the new state and are so correctly controlled, previous existing ones may be impacted. Indeed, when the desired next hop does not exist anymore or if its index has changed, the resulting path will change. In CFLB, the impacted controlled flows (i.e., the elephants ones) will fall back to a random load balancing and so will be distributed similarly to classic hash-based load balancing. We consider that such topological changes should be quite marginal (at a time scale greater than flows duration) and new subflows may be created if impacted ones share a common bottleneck.

5.4.4 Avoiding polarization

Some hash-based load-balancing techniques suffer from the polarization problem [MMH07]. This problem arises when there are several load-balancing routers in sequence. If they all perform the same computation on the received packets, they will select the same next hop resulting in an uneven traffic distribution. With CFLB, the polarization problem only arises when packets traverse more than L CFLB routers. Every router spaced by L hops computes the same next-hop selection. CFLB solves this problem by assuming that the every packet from the same flow received on a router has the same TTL[4]. CFLB therefore includes the TTL of the packet inside the hash function ensuring that all routers will make different next-hop selections along the path[5]. Respectively Eq. 5.5 and Eq. 5.7 become:

$$E_i(n_i) = (n_i + H(u_f || ttl_i)) \mod B \tag{5.10}$$

$$D_i(p) = (\left\lfloor \frac{p}{B^i} \right\rfloor - H(u_f || ttl_i)) \mod B \tag{5.11}$$

ttl_i is the TTL of the packet received at the i^{th} router on the path.

5.4.5 Summary

In the previous section, we have explained all the design decisions behind the CFLB algorithm. For clarity, we provide in this section the detailed pseudocode of CFLB.

[4]This is a reasonable assumption since hosts use the same TTL for all packets and all packets from a flow follow the same path.

[5]Another solution could have been to use a simple router id as in classical hash-based load-balancing techniques [MMH07], however this requires each router to be configured with a unique router id and requires the sources or the network information server to know all router ids.

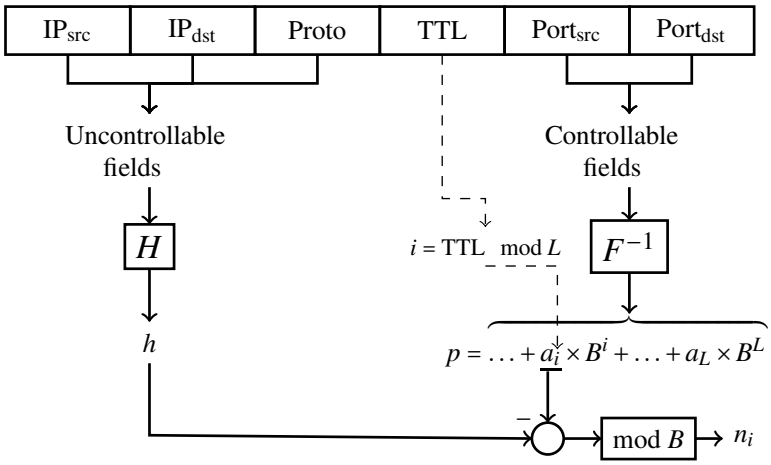


Figure 5.8: The complete mode of operation of a CFLB router.

Figure 5.8 and Algorithm 5.1 show respectively the operations and the pseudocode performed by a CFLB router to forward a packet among load-balanced next hops. The first operation is to extract the controllable and the uncontrollable fields and the TTL from the packet header. Operation F^{-1} is the inverse of the invertible function that extracts the path selector from the controllable fields. In Algorithm 5.1, after having retrieved the next-hop selection n_i, the router performs the operation introduced in Section 5.4.3, i.e., an if-then-else on the value n_i, to determine whether the packet was controlled by the source. The a_i value, in Figure 5.8, corresponds to the addition modulo B of the next-hop selection performed at position i and the hash computed on the uncontrollable fields. This a_i value was inserted by the source at the i^{th} position inside the path selector. The router can thus retrieve it and compute the subtraction modulo B with the same hash value, to finally retrieve the next-hop selection n_i.

Algorithm 5.2 shows the pseudocode used by sources to construct a path selector. The source needs to first compute the length of the path selector. This is needed because the routers position themselves inside the path selector by using the TTL and the length of the path selector. Thus, this length has to be taken into consideration to find the path selector. Then, the source iterates over all routers on the path (with a TTL ttl_i and a next-hop selection n_i) to compute the path selector.

Network-wide constant: B = The radix in use in the network.
Network-wide constant: X = The number of bits that are controllable in the packet header.
Require: *pckt* = The packet to forward.
Ensure: The next-hop selection to apply on *pckt*.

1: $L \leftarrow \lfloor \log_B(2^X) \rfloor$
2: $c_f \leftarrow$ `ExtractControllableFields`($pckt$)
3: $u_f \leftarrow$ `ExtractUncontrollableFields`($pckt$)
4: $ttl \leftarrow$ `ExtractTTL`($pckt$)
5: $p \leftarrow F^{-1}(c_f)$
6: $n_i \leftarrow (\lfloor \frac{p}{B^{(ttl \mod L)}} \rfloor - H(u_f \| ttl)) \mod B$
7: **if** $n_i < N_k$ **then**
8: **return** n_i
9: **else**
10: **return** $H(u_f \| c_f \| Router_{ID}) \mod N_k$
11: **end if**

Algoritm 5.1: Pseudocode showing operations performed by a CFLB router.

Network-wide constant: B = The radix in use in the network.
Network-wide constant: X = The number of bits that are controllable in the packet header.
Require: *path* = A sequence (ttl_i, n_i), where ttl_i is the TTL of the packet when received by router i and n_i the next-hop that should be selected by router i.
Require: u_f = The uncontrollable fields the source needs to use.
Ensure: The controllable fields (c_f) to use to force a packet to follow the load balanced path *path*.

1: $L \leftarrow \lfloor \log_B(2^X) \rfloor$
2: $p \leftarrow 0$
3: **for** $(ttl_i, n_i) \in path$ **do**
4: $p \leftarrow p + ((n_i + H(u_f \| ttl_i)) \mod B) \times B^{(ttl_i \mod L)}$
5: **end for**
6: **return** F(p)

Algoritm 5.2: Pseudocode showing the path selector construction.

5.5 Evaluation

In this section, we evaluate the performance of CFLB compared to traditional hash-based load-balancing. We use the source and destination port numbers as controllable fields and the IP addresses are used as the uncontrollable fields for CFLB. Our goal is twofold: first, evaluate its load-balancing

and forwarding performances, and second, show through simulations and experiments how Multipath TCP can benefit from CFLB to exploit the underlying path diversity.

5.5.1 Performance evaluation

Load balancing for non-controlled flows

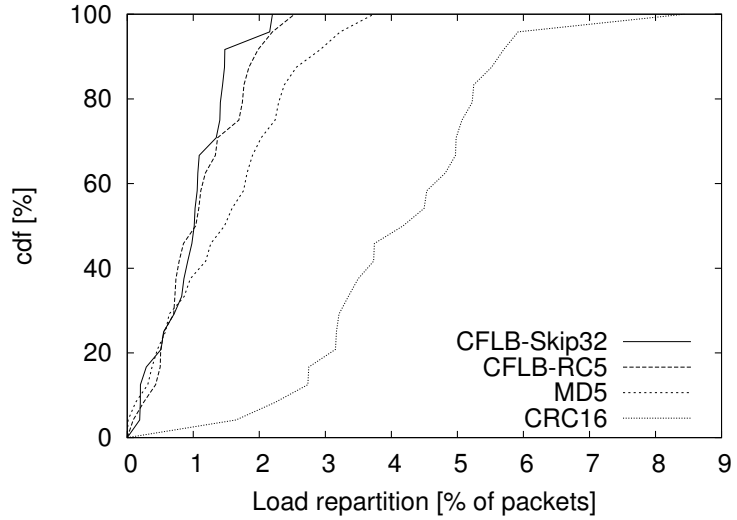
We first evaluate whether a router having multiple next hops for a given destination uniformly distributes the load [CWZ00] for non-controlled flows. If there exist N next hops for a given destination prefix, the load balancer should distribute $\frac{1}{N}$ of the total traffic to each next hop.

CFLB enables sources to steer controlled packets while also acting as a classic load balancer for non-controlled packets (mice flows). To compare the hash-based load balancing techniques and CFLB, we simulated each method using realistic traces and evaluated the fraction of the packets forwarded to each next hop. We based our simulations on the CAIDA passive traces collected in July 2008 at an Equinix data center in San Jose, CA [SACA].

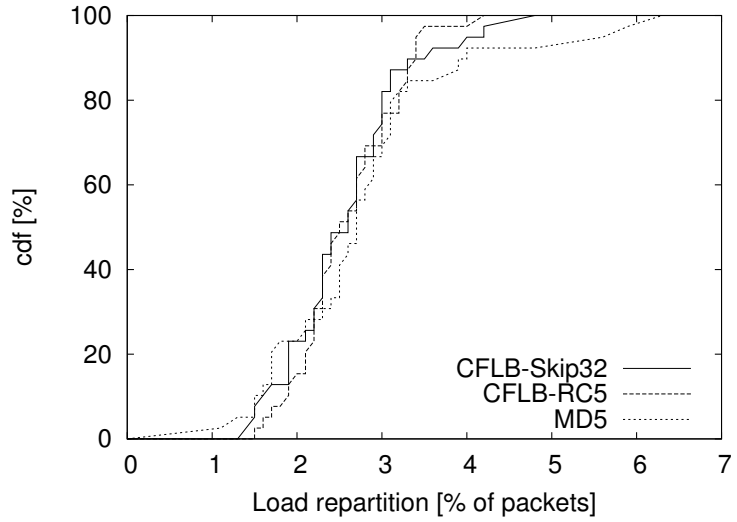
To analyze how CFLB balances the non-controlled traffic compared to hash-based techniques, we first simulated 10 million packets (extracted from the CAIDA traces) forwarded through one load balancer performing a distribution among $N = 2$ next hops. Figure 5.9(a) shows the result of this simulation (computed every second). There are three observations resulting from Figure 5.9(a). First, using CRC16 as a hash-based load balancer gives a rather poor distribution of packets. Second, as the maximum deviation value never goes up to 4% of packets, the load distribution among the two output links is close to an equal 50/50 % repartition of traffic for all evaluated techniques except CRC16. Third, CFLB, whatever the block cipher used, achieves an equivalent load distribution as a hash-based load balancer using MD5. We did not observe a significant impact on the quality of the load distribution according to the value of B used.

We also evaluated the load balancing performance considering a sequence of several load balancers. Figure 5.9(b) shows the cumulative distribution of the maximum deviation of the load distribution after crossing four subsequent load balancers (computed every half second). The same observation as for Figure 5.9(a) applies, CFLB performs at least as good as a classical hash-based load balancing technique.

Another performance factor is how the load distribution varies over time. Figure 5.10(a) and Figure 5.10(b) show, for respectively a hash-based



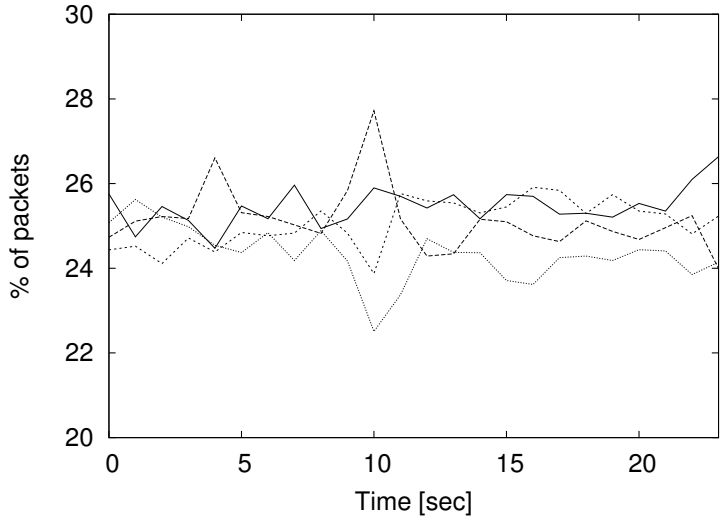
(a) After one load balancer.



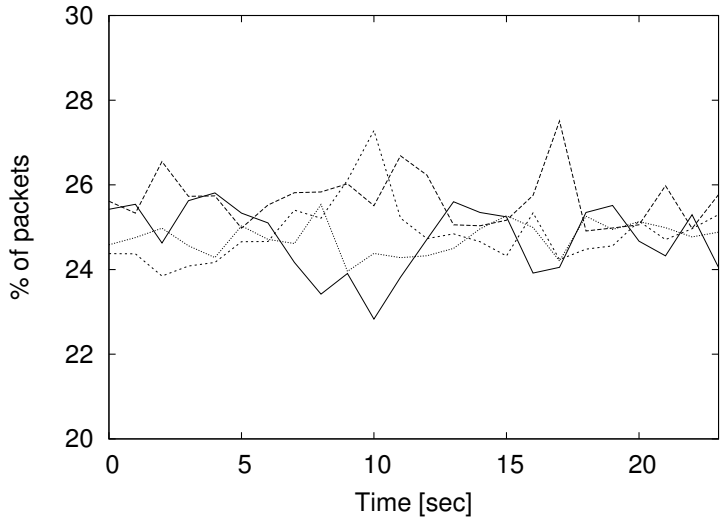
(b) After 4 load balancers.

Figure 5.9: Deviation from an optimal distribution amongst two possible next-hops.

load balancer using MD5 and CFLB using RC5, the load balancing distribution of packets over time when $N = 4$. We analyze here the case of a router having four outgoing links toward a given destination. A perfect load balancer would send 25% of the packets on each link. We can notice that there are no significant differences between the two techniques, as they behave in the same way over time. They both slightly fluctuate within the same tight interval $[22\%, 28\%]$ and their median is close to 25%. Simulations with other traces and different values of N provide similar results.



(a) MD5 Hash-Based



(b) RC5 CFLB

Figure 5.10: Packet distribution computed every second amongst four possible next hops.

Forwarding performances

The second requirement is the forwarding performance. In order to evaluate it, we implemented the forwarding path of CFLB as a module in the *Linux kernel 2.6.38*[6]. The basic behavior of the Linux kernel when dealing with multiple next hops for a given destination is to apply a round robin

[6]More information can be found at: `http://inl.info.ucl.ac.be/cflb`.

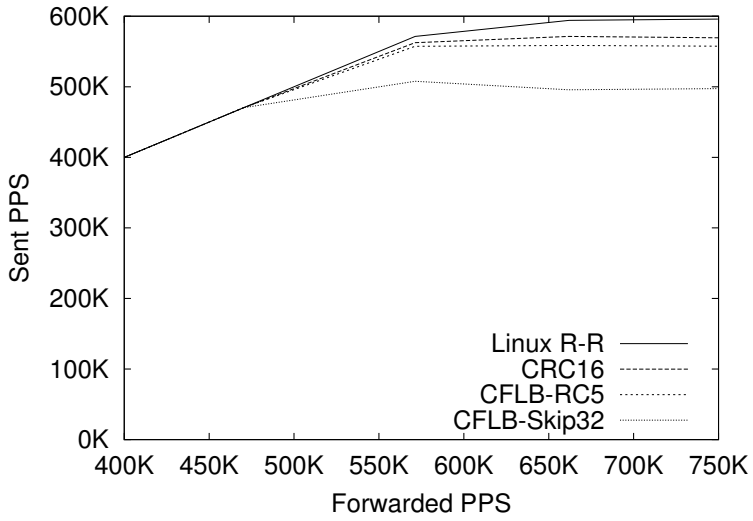


Figure 5.11: CFLB gives equivalent forwarding performance as hash-based load balancers.

distribution of packets based on their IP addresses, therefore performing a pure layer-3 load balancing. As this is not comparable to the hash-based load-balancing behavior introduced in Section 5.2.1, we extended the Linux kernel to take into consideration the 5-tuple of the packets and then apply a hash function to select a next-hop (only CRC-like functions are available). To implement CFLB in the kernel, we extend the previously mentioned hash function to enable the deterministic selection of a next hop as described in Section 5.4. We used two different 32-bit block ciphers to implement the invertible function: RC5 and Skip32. The implementation of these two block ciphers has not been optimized, the goal is solely to prove the feasibility of our solution (various techniques could be used to improve its performance [KKG+10, HJPM10]). Note that CFLB also computes a CRC function over the uncontrollable fields to add randomness for non-controlled flows.

We deploy a testbed of three computers to evaluate the performance of the forwarding path of a Linux router. The computer acting as a load balancer is an Intel Xeon X3440 @2.53GHz, and both sender and receiver are AMD Opteron 6128 @2GHz. The sender is connected through a 1Gbps link to the load balancer which balances traffic amongst two 1Gbps links to the receiver. The traffic was generated using 8 parallel *iperf* [ipe13] generators, creating UDP-packets with a payload of 64 Bytes, in order to overload the load balancer. The result of this experiment is given in Figure 5.11.

The classic Linux Round-Robin on the IP-addresses obviously performs the best (it only requires a lookup of the destination IP address in the

routing cache to forward the packet). It forwards approximatively 600,000 packets per second. Not far below, both the classical hash-based technique using CRC16 and CFLB using RC5 are able to forward respectively 570,000 and 560,000 packets per seconds. This performance decrease compared to the standard Linux Round-Robin is mainly due to the more complex hash-algorithm that selects the next hop. Finally, CFLB using Skip32 forwards up to 500,000 packets per second. We can conclude that CFLB, even using non-optimized block ciphers, comes with a marginal cost.

5.5.2 MPTCP improvements with CFLB

In the following section, we evaluate the advantages of using MPTCP hosts conjointly with a CFLB-enabled network. With CFLB, Multipath TCP is guaranteed to efficiently use the network resources as subflows can be deterministically mapped to paths. We first simulate a data center environment to show that when using CFLB, MPTCP requires fewer subflows for elephant connections than with a probabilistic approach. Finally, we also show in a small testbed that MPTCP can benefit from the usage of CFLB to avoid crossing hot spots.

Data center simulations

We performed simulations of MPTCP-enabled data centers and evaluate the performances achieved when a simple central flow scheduling algorithm allocates elephant flows. The scheduler that we use for simulations simply consists in minimizing the number of flows going through each link of the data center. In practice, hosts can use a similar technique as in [CKY11] to detect whether one connection corresponds to an elephant flow, and if so query the scheduler to establish additional subflows. The scheduler then specifies to the host the TCP ports to be used to setup a new subflow. The required TCP ports are computed using the CFLB mode of operation allowing to map a subflow to a specific path in the network. We refer to this combination of Multipath TCP and CFLB in the remaining of this section as MPTCP-CFLB.

To evaluate the benefits of CFLB with MPTCP in data centers, we first enhanced the *htsim* packet-level simulator used in [RBP+11] to support path selection with CFLB. We consider exactly the same Fat-Tree datacenter topology as discussed in Figure 2 of [RBP+11]. This simulated datacenter has 128 MPTCP servers, 80 eight-port switches and uses 100 Mbps links. The traffic pattern is a permutation matrix, meaning that senders and destinations are chosen at random with the constraint that destinations do not sink more than one connection. The regular MPTCP bars of Figure 5.12

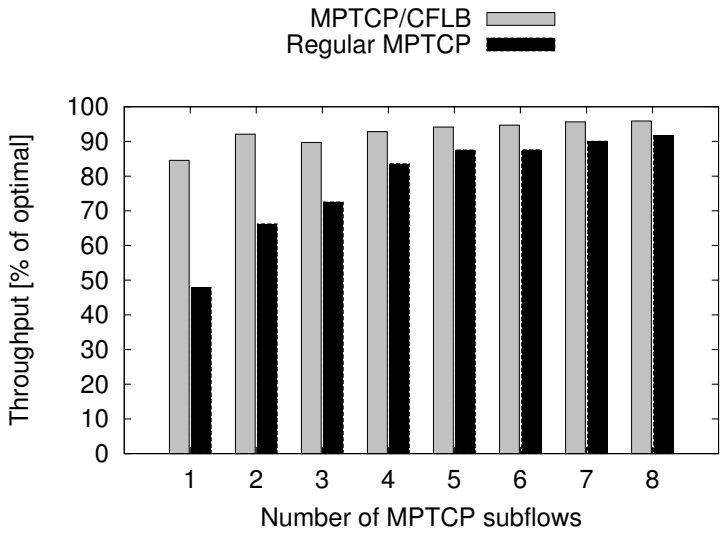


Figure 5.12: MPTCP needs few subflows to get a good Fat Tree utilization when using CFLB.

are the same as Figure 2 of [RBP+11]. It shows the throughput achieved by MPTCP when MPTCP subflows are load-balanced using ECMP. The MPTCP-CFLB bars show the throughput that MPTCP is able to obtain when CFLB balances the MPTCP subflows over the less loaded paths. The simulations show that with only 2 subflows, MPTCP-CFLB is much closer to the optimum than MPTCP with hash-based load balancing. Even with only one subflow (smartly allocated by the scheduler), improvements are considerable and MPTCP-CFLB achieves a good utilization of the network. This can be explained by the fact that relying on a random distribution of subflows ends in a poor use of the available resources.

Similar results have been observed on other data center topologies such as VL2 and BCube. We also performed simulations for an overloaded data center and observed that using MPTCP-CFLB conjointly with a flow scheduler focusing on less congested paths offers more fairness amongst the different connections.

Testbed experiments

We modified the MPTCP *Linux kernel 2.6.36* implementation [BPB11] to add the deterministic selection feature offered by CFLB. We created a *netlink* interface to the kernel so that a user-space module can interact with MPTCP and announce to the kernel the subflows to create. The CFLB functionality has been implemented in user-space. Our prototype allows to control the source and destination TCP ports to follow a specific path

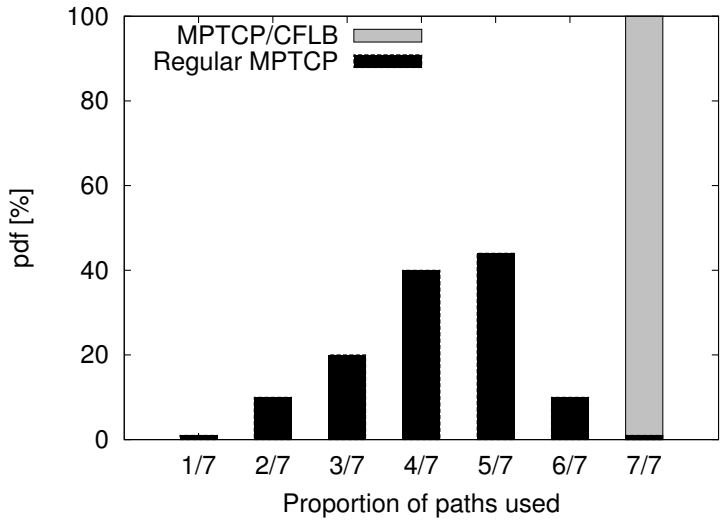


Figure 5.13: Regular MPTCP is unlikely to use all paths. MPTCP-CFLB on the other hand always manages to use all the paths.

inside an IPv4 network. The two block ciphers (RC5 and Skip32) were also implemented inside the kernel crypto library.

A python library *pycflb* was developed to provide a simple API for interacting with the user-space CFLB. We also developed an RPC server to show the feasibility to centralize the computation of the paths. This server has information about the network topology and is the only one that interacts with the pycflb library. pycflb can be configured with the cipher and key parameters in use in the network. Sources only query it to retrieve the ports to use or to recover the path taken by a specific flow. These three implementations (Linux MPTCP netlink-interface, user-space CFLB and pycflb library) allow a source to deterministically map subflows to paths and represent approximatively 4,000 lines of code.

When Multipath TCP runs on a single homed server, additional subflows are created by modifying the port numbers in a random manner. Since Multipath TCP relies on tokens to identify to which MPTCP connection a new subflow belongs, both the source and destination TCP ports can be used to add entropy. Combining CFLB and Multipath TCP in the Linux MPTCP implementation provides a significant benefit because the subflow 5-tuple can be selected in such a way that the underlying path diversity offered by the network can be easily exploited. We evaluate the benefits of this technique in a small testbed with a client and a server (AMD Opteron 6128 @2GHz) and two CFLB-capable routers (Xeon X3440 @2.53GHz).

In the first experiment, each host is connected to one router via a 1Gbps

link. The routers are directly connected via seven 100 Mbps links. These 7 links offer 7 different distinct paths between the client and the server. If seven MPTCP-subflows are created, an optimal usage of the network should result in about 700 Mbps of throughput. To evaluate this, we ran iperf between the hosts, creating traffic during one minute. The experiment has been repeated 400 times to collect representative results. Figure 5.13 provides the probability distribution function of the number of distinct paths used by the classical MPTCP and our enhanced MPTCP-CFLB implementation.

Figure 5.13 raises the following observation: as expected, using seven subflows, MPTCP-CFLB is able to take the full benefit of the seven paths while the classical Multipath TCP cannot efficiently utilize them. Indeed, the performance of MPTCP-CFLB is completely deterministic as the MPTCP connection balances exactly its seven subflows over the seven paths. Among the 400 experiments, when paths are randomly selected, only two experiments were able to use the seven paths. This can be confirmed by the covering probability function defined in Equation 5.1. Therefore, covering the whole 7 load-balanced paths with 7 subflows the probability is $P^{7}_{(7,7)} = 0.6\% \approx \frac{2}{400}$ which explains the poor result of regular MPTCP. Most of the experiments result in four or five paths being used. This implies that two or three paths carry two competing TCP subflows from the same MPTCP connection.

Our second evaluation (see Figure 5.14) still offers 7 distinct paths from the source to the destination, but this time the destination has two 100 Mbps links. One is a direct link from the load balancer to the destination and the second is attached to the router. With only two subflows, MPTCP-CFLB is able to saturate the two 100 Mbps interfaces of the destination. Figure 5.15 compares the performance of MPTCP and MPTCP-CFLB when the number of subflows varies. Each measurement with MPTCP was repeated 100 times and Figure 5.15 provides the average measured goodput. These measurements clearly show that when using random TCP port numbers, MPTCP is unable to efficiently use the two different 100 Mbps links. Increasing the number of subflows slowly increases the performance, but adding a subflow to an MPTCP connection comes with a significant cost. Thus, the less TCP sublows are established, the best it is. MPTCP-CFLB is able to cover all the available paths with the minimal cost.

5.6 Deployment and applications

In this chapter, we have mainly focused on the utilization of CFLB in data centers networks carrying TCP/IPv4 packets. CFLB could be applied to other networking technologies. A first natural extension of CFLB

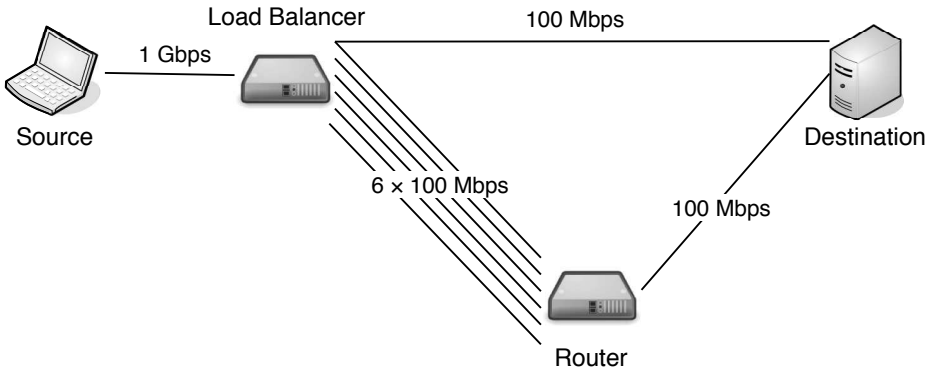


Figure 5.14: Testbed – The maximum throughput available between S and D is at 200 Mbps due to the bottleneck link between the router and the destination.

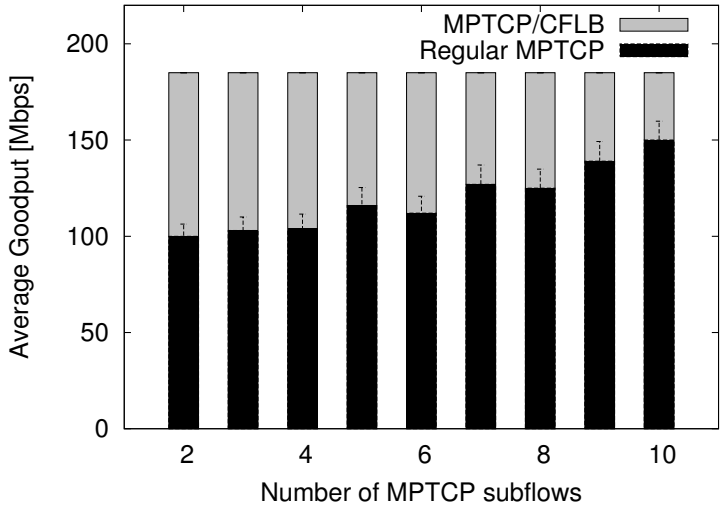


Figure 5.15: Regular MPTCP has a very small probability of using link A of Figure 5.14 and is thus suboptimal compared to MPTCP-CFLB.

would be to deploy it for monitoring purposes. Monitoring load-balanced paths depends on packet steering and is thus difficult. Being able to monitor an ISP network or a data center network would certainly help network operators.

In the rest of this section, we look at how the generic design of CFLB can be adapted to real network environments. We first describe how CFLB can be adapted to different networking technologies. Then, we describe several applications that could exploit the path steering capability of CFLB.

Finally, we provide some examples to setup CFLB based on the network graph characteristics and discuss its limitations.

5.6.1 Deployment into current networking technologies

CFLB can be used with various network technologies. Depending on the technologies, the controllable and uncontrollable fields of the packets header may differ. We provide here a non-exhaustive list of examples where CFLB can be easily deployed.

IP version 4

In traditional ECMP, the load balancing decision is taken on the basis of the source and destination IP addresses, the protocol (UDP or TCP), and the source and destination ports. They form what is called the *5-tuple*. In CFLB this 5-tuple is divided in controllable and uncontrollable fields. Our recommendation to support CFLB in IPv4 is to use the source and destination transport ports as controllable fields. This implies that the path selector is encoded as a 32-bit field, we call this basic version CFLB-32.

Moreover, two extensions are possible. First, for ISP networks where monitoring applications need to monitor very large paths, it is possible to extend the number of bits in controllable fields. In MPLS networks, some monitoring applications use `127.0.0.0/8` as the source address for monitoring packets [KS06]. Using the same approach, it would be possible to use 24 bits from the source address as part of the controllable fields, bringing a total of 56 controllable bits (CFLB-56). Second, data centers often use homogeneous servers and specialized network stacks. Although IPv4 supports fragmentation, almost all TCP/IP stacks use Path MTU discovery [MD90] and avoid fragmentation. With such a recent stack, the 16 bits of the packet identification are not required anymore and could be included in the controllable fields. This would bring 48 controllable bits (CFLB-48).

IP version 6

The IPv6 header provides a larger number of bits that can be controlled by CFLB. With IPv6, the basic solution is to use the source and destination ports as controllable fields. The 20-bit flow label field of the IPv6 header [RCCD04] could also be used as a controllable field. Although there have been discussions on using the IPv6 flow label field for load balancing [CA11], the status of this field is still unclear today. For monitoring applications, since the 64 least significant bits of the destination address are not actually

used for routing [dVPC+08], they could be part of the controllable fields. For some applications, the low order 64 bits of the source address can also be controlled. With the source and destination ports, this would make a total of 160 controllable bits.

MultiProtocol Label Switching (MPLS)

MPLS is widely used in large ISP networks notably to provide Virtual Private Network services. *Label Switching Routers* (LSRs) exchange labelled packets over *Label Switched Paths* (LSPs) [RVC01]. Each packet contains a stack of 32-bit headers and each header contains a 20-bit label and a Time-To-Live. LSPs can be established by using different control plane protocols. The *Label Distribution Protocol* (LDP) [AMT07] uses the same paths as the IGP routing protocol while RSVP-TE [ABG+01] is able to establish an LSPs that meets traffic engineering constraints. Two types of load balancing are possible in MPLS networks. First, LSPs established by using LDP use the same paths as those provided by the intradomain routing protocol. Load balancing will occur when several paths have the same cost, which is frequent as indicated by our analysis in Section 5.2.2. Second, RSVP-TE establishes a single path between the ingress LSR and the egress LSR and there is thus no load balancing along one LSP. However, a recent extension to RSVP-TE proposes to automate the establishment of several parallel traffic engineered LSPs [KH13] between a pair of LSRs. In this case, load balancing needs to be performed by the ingress LSR.

CFLB can be implemented in several ways in MPLS networks. If the network carries IPv4 or IPv6 packets, LSRs can use fields from the packet header for load balancing. Some existing LSRs already support this form of load balancing [BFD+11]. Note that the top label of the MPLS header cannot be used to perform load balancing as the label changes on each hop. However, as MPLS is becoming a generic packet transport technology, MPLS packets often carry non-IP packets (e.g., ATM, frame-relay or Ethernet frames). In this case, these packets can be encapsulated by using an additional header such as those proposed in [BFD+11] and [KDA+12]. These two techniques allow the ingress LSR to control per-flow bits in this additional header consecutive to the MPLS top one. This per-flow information can be used as input by CFLB.

Transparent Interconnection of Lots of Links (TRILL)

TRILL is an IETF working group that finalizes a solution to enable shortest-path frame routing in multi-hop Ethernet networks [TP09]. TRILL is based on the RBridges [Per04] initially proposed by Radia Perlman. It

replaces the IEEE 802.1d spanning tree protocol by a variant of the IS-IS routing protocol and defines a new frame header that is used to forward Ethernet frames. The header contains a hop-count field that could be used by CFLB and existing TRILL switches which already support load balancing [CS13b]. For datacenter applications running IPv4 or IPv6 over Ethernet, CFLB implementation on RBridges would use the hop-count from the TRILL header and the fields of the IP header. For monitoring applications that do not run above IP, one possibility would be to use a multicast MAC address as source.

5.6.2 Applications enabled by CFLB

In this section, we describe several applications that can benefit from the ability of CFLB to control the path forwarding on ECMP. We first analyze monitoring applications that are typically used on routers, then tunneling applications and discuss how multihomed hosts could exploit CFLB.

CFLB-enabled traceroute

To diagnose routing problems, network operators often use `traceroute` which is unable to correctly report paths in a network layer load balancing environment (see Section 5.2.3). *Paris traceroute* fixes the issues of `traceroute` in that respect, and provides a probabilist algorithm to discover the set of equal cost multipath routes between a pair of routers. The algorithm evaluates the number of probes to send in order to discover all routes with a given bound on the failure probability. The probes use somehow random source ports that are expected to be distributed over all possible paths. Because of its statistical nature, Paris traceroute is likely to issue a prohibitive number of probes to achieve a low failure probability in complex ECMP cases. Moreover, transient failures may impact the behavior of the algorithm, leading to uncertain results.

In networks using CFLB, `traceroute` can be extended to take as input a path selector and the CFLB parameters (F, H, and B) in order to send probes along a specific path. Moreover, `traceroute` can be be coupled with an algorithm such as Algo. 5.3 to be able to discover *all* multipath routes between a source and a destination without knowing the topology (assuming that the next hops are incrementally sorted). The algorithm defines the procedure `Discover` which for each hop incrementally probes the available next hops, until the same next hop is found again. The procedure `Probe`(p, ttl) computes an algorithm similar to Algorithm 5.2 that takes a path selector p and computes the controllable fields c_f to generate and send a probe along the desired path. It eventually returns the address of the router

Procedure: `Discover`(p, ttl, V)
Require: p, a path selector
Require: B, the max wide constant $\#nexthops$
Require: ttl, the TTL to use in probes
Require: V, the set of routers already reached
1: $R \leftarrow \varnothing$
2: $p' \leftarrow p$
3: **while** *true* **do**
4: $p' \leftarrow p' + B^{ttl}$
5: $n \leftarrow$ `Probe`(p', ttl)
6: **if** $n \in R$ **then**
7: **break**
8: **end if**
9: **if** $n \notin V$ **then**
10: `Discover`$(p', ttl + 1, V \cup \{n\})$
11: **end if**
12: $R \leftarrow R \cup \{n\}$
13: **end while**

Algoritm 5.3: Discovering an CFLB load balancing DAG.

that replied with an ICMP message. We make the assumption that routers always send ICMP messages back with the same source address.

We compare the discovery capability of Paris traceroute to the usage of CFLB in a simple environment of successive ECMP-enabled routers. Unfortunately, in the current Paris traceroute implementation, the failure probability cannot be adjusted, so we use the default setting. The test bed topology works as follows (see Figure 5.16), the routers on the primary path between the source and the destination use ECMP with CRC-16 over the 5-tuple with a seed to avoid polarization. Each of them has another path to the destination in the first experiment (as in Figure 5.16), or three others

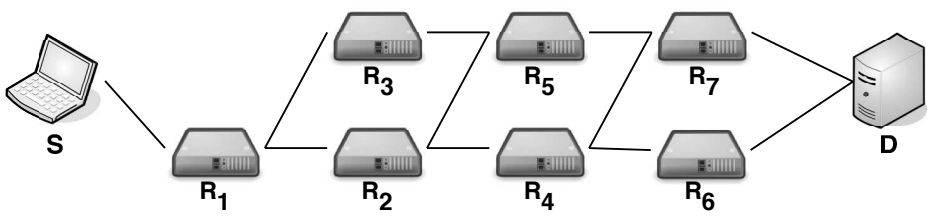


Figure 5.16: Simple topology where 3 routers are CFLB-enabled (R_1,R_2 and R_4) and have 2 possible next hop to join destination D.

in the second experiment. All paths from a given router to the destination are of equal length. While the first hops are easy to discover using the probabilistic approach of Paris traceroute, the last ones are more difficult to detect since the set of flow identifiers decreases along each path. We vary the length of the paths between three and eleven, and thus the number of ECMP-enabled routers between one and eight. We observe that Paris traceroute discovers all hops when there are up to 6 ECMP-enabled routers with two choices, using 1492 probes, but only up to three routers when there are four choices, using 296 probes. In comparison, our implementation of CFLB coupled with the discovery algorithm requires only 34 and 29 probes, respectively. Our improved CFLB-`traceroute` implementation allows to quickly discover the topology for applications that need routing information to select their paths. Thanks to its deterministic nature, CFLB does not rely on any failure probability setup.

Congruent Failure Detection

Bidirectional Forwarding Detection (BFD) [KW10] is a failure detection protocol that complements the detection components of various protocols. BFD can be used as a companion to routing protocols like OSPF or BGP to quickly detect link failures. BFD monitors the connectivity between a pair of systems by using different techniques including the periodic transmission of *Echo* packets. However, the utilization of ECMP paths raises some problems. Consider for example the utilization of BFD to detect the possible failure of an iBGP session through a network that offers multiple paths. A classical ECMP load-balancing technique would probably forward the BFD and BGP packets over different paths and BFD would not be able to detect the failures that affect the iBGP session.

Using CFLB, a different approach is possible. Routers know the network topology from their link state database and can easily compute all the equal cost paths towards a given destination. With this information, the router can easily find the source and destination ports required to force the BFD packets to follow the same path as the monitored iBGP session. In practice, the two endpoints of the BGP session would need to agree on the source and destination ports that each router uses for monitoring purposes.

Monitorable tunnels

As another example of the benefits of CFLB, consider a network that provides a VPN service and uses several tunnels for redundancy between

each pair of *Provider Edge* (PE) routers. Such a configuration is for example used by the AMS-IX Internet exchange[7]. In such network, the client connected to the PE router could be interested in being able to monitor the quality of each VPN tunnel. If the PE routers load balance the VPN packets by using a classical hash-based technique, then it is very difficult for the client to discover the different paths and monitor them separately. The only possible solution is to statistically try to explore some paths. With CFLB, the VPN provider could either supply to its client the CFLB parameters used on the PE router for load balancing or supply the 5-tuples it needs to use for monitoring all paths.

SLA verification

The verification of *Service Level Agreements* (SLA) is an important monitoring problem in many commercial ISPs. Commercial tools running on routers exchange monitoring packets to measure these SLAs and report violations [CS13c]. With ECMP the equal-cost multiple paths may not all meet these SLAs [MC09]. With CFLB, the sending router can compute a path selector that will allow to probe a particular path or a set of paths to verify the SLAs.

5.6.3 Calibration and limitations

CFLB is easily parametrizable when knowing the network routing characteristics. The values of B and the number of controllable bits might vary according to the environments and applications. If the maximum number of ECMP next hops in the network is small, CFLB can use a small B value increasing therefore the length of the paths selector, i.e., the number of controllable routers.

Figure 5.17 shows for different values of X, i.e., the number of controllable bits, the number of router's decisions (see Equation 5.4) that can be encoded in p regarding the value of the radix B. If we consider a worst case scenario, where there exists a path whose length is equal to the diameter of the network and where each node potentially offers maximal load-balancing capabilities. We observed that there are at most 6 next hops on each router for our set of ISP networks presented in Section 5.2.2 (see Figure 5.1). With $B = 6$ and CFLB-32, based on Figure 5.17, one is able to encode 12 router decisions inside the path selector.

We found that, for all the ISP networks evaluated in Section 5.2.2, the maximum length of each path is lower than 16 hops. In the case of CFLB-

[7] See `http://www.ams-ix.net/infrastructure-detail/`.

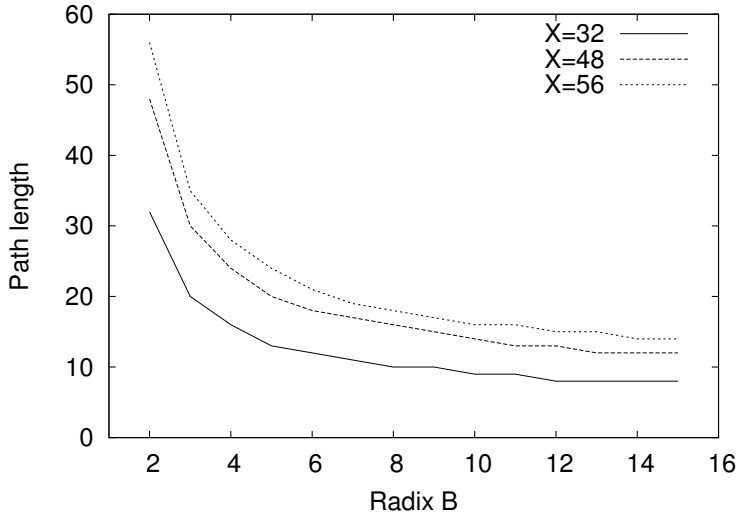


Figure 5.17: The large B value is the less routers decisions can be encoded inside p.

32, there are not enough bits to encode a path of 16 hops. Only 12 hops long path can be encoded. However, depending on the location of CFLB-enabled routers on the paths, one can encode a path longer than 12 hops in the path selector. Indeed, if one or more router within the 12 hops does not offer local diversity towards the destination, then, one can encode in the corresponding part of the path selector the decision for the router located 12 hops later. If one of the 6 first hops of longest paths does not provide multi-next hops, we can re-use the forwarding decision of the i^th to encode the $12+i^{th}one$. To understand how efficient can be this technique in practice within ISP topologies, we evaluate the number of path that contains 2 or more ECMP-enabled routers separated by 12 hops, i.e., the number of non entirely controllable paths. Results showed that only a very small number of equal cost paths are impacted by the issue, only 0.5%, 0%, 0.05% and 0.03% respectively for Tiscali, AboveNet, Tier-1 ISP 1 and Tier-2 ISP 2. This limitation is negligible for multipath transport applications on servers but may impact monitoring applications. To fully explore all paths, one needs to increase the path selector size or use additional monitoring sources. From Equation 5.4 we can derive:

$$X = \lceil \log_2(B^L) \rceil \tag{5.12}$$

To cover networks that are 16 hops wide and with maximum 6 potential next hops 42 bits are therefore required. In this extreme case, the use of CFLB-48 ore CFLB-56 are mandatory (see Section 5.6.1).

In Section 5.2.2, we saw that the ECMP local diversity of data center

is high. We found that at maximum there are 10 next hops for the BCube topology. Therefore, a minimal value of $B = 10$ is required to represent the routers forwarding decision. Compared to ISPs topologies, paths in data center contain a small number of hops, the longest shortest path only has 6 hops. Thus, 20 bits are sufficient to encode the path selector, this is lower than the number of controllable bits CFLB can use in a data center environment (48 bits for IPv4). Furthermore, these environments are particularly well suited for multipath transport layer applications on the server-side which do not require a complete end-to-end path exploration.

5.7 Conclusion

Most data centers networks rely on hash-based load balancing to distribute the load over multiple paths. Hash-based techniques allow to efficiently spread the load but it is difficult to predict and force the next-hop selection that such load balancers will take. In this chapter, we have shown that it is possible to achieve both efficient load balancing while enabling hosts to explicitly select the paths of their flows without storing any state in the network. This opens new ways for MPTCP-enabled hosts and load balancers to interact in order to improve the network performance and utilization.

Controllable per-Flow Load Balancing relies on invertible functions, such as block ciphers, instead of solely using classical hash-based selection. Thanks to an invertible operation, it is possible to preserve the properties of currently used hash functions with the added benefit of enabling CFLB-aware sources to steer elephant flows over selected load-balanced paths. We envision a data center where mice flows are distributed using the classical load balancing model while elephant flows are deterministically allocated to less congested paths. Our simulations indicate that CFLB is as efficient as classical hash-based techniques to achieve an optimal load distribution. Performance measurements in our lab have shown that our prototype implementation in the Linux kernel achieves almost the same performance as the default load balancing. Furthermore, we have shown that by coupling CFLB with MPTCP it is possible to greatly improve the utilization of data center networks offering path diversity between pairs of servers.

We hope that CFLB will encourage other researchers and network manufacturers to reconsider the utilization of blind hash functions in various types of load balancers.

Chapter 6

Conclusion

The objective of this thesis was to evaluate the deployment and advantages of Multipath TCP in various types of networks. We have shown that Multipath TCP can bring advantages inside data centers but also and more importantly for mobile devices. Indeed, the ubiquity of WiFi Access Points and the cost of mobile data network connectivity encourages the smartphone users and network operator to use WiFi instead of 3G. Moving from 3G to WiFi is not currently supported by the Internet protocol suite burdening the end user and decreasing its overall experience. We showed that MPTCP provides an actual support for handovers for these users as their connections can survive connectivity losses. We finally showed that end users can benefit from MPTCP in today's Internet with legacy servers thanks to the use of protocol converters.

Thanks to its many advantages, MPTCP will soon play an important role for mobile users and manufacturers as well as alternative to other – more expensive – mobile data solutions offloading from 3G or LTE to WiFi. *Apple Inc.* is the first smartphone manufacturer to believe in MPTCP [Bon13]. It uses MPTCP in a specific application (Siri) to reduce the delay and improve the reliability of the data stream sent by the smartphone [app14]. If Apple believes in MPTCP, it is very likely that other big players will follow.

6.1 Detailed contribution

The contributions of this thesis are threefold. First, we implemented the mobility support of MPTCP and evaluated its benefit in a mobile environment. Second, we analyzed the deployability of MPTCP by developing a tool to identify middleboxes that can interfere with the TCP connections. We also have analyzed the deployment of MPTCP regarding the incentive

problem and have proposed a solution to accelerate its deployment. Finally, we showed that data center networks can be improved to better support multipath protocols.

In Chapter 2, we presented different ways of performing mobility in a WiFi/3G environment with MPTCP depending on the user requirement, i.e., if he desires high bandwidth, reduce cost and battery lifetime. Our experiments in commercial networks show that MPTCP could play a role to improve mobile users' experience as well as for WiFi/3G convergence. Compared to custom applications that are difficult to realize in practice, MPTCP reacts quickly to recover from WiFi losses in presence of 3G with only a small impact on the application delay and goodput.

In Chapter 3, we present `tracebox`, a new extension to the well known `traceroute` tool. `tracebox` can detect middleboxes interference by generating probes with increased TTL values and compare the ICMP replies with the probes sent. `tracebox` gives a flexible interface to generate a large space of different types of probes. We learned from this part that there exists middleboxes that remove MPTCP options as it is often considered as an unknown TCP option by middleboxes. Nevertheless, we found that these middleboxes are not common which implies that MPTCP-enable devices can most of the time benefit from MPTCP.

Chapter 2 motivated Chapter 4. Indeed, we showed that MPTCP brings many benefits to the end user. However as every new protocol it suffers from deployment incentive problem. Both hosts have to support it to benefit from it and there is no incentive for one to support it if the other one does not. We proposed to place on the Internet protocol converters that convert MPTCP connections to regular TCP. Allowing the host that redirects its connection through the converter to benefit from MPTCP. We showed that it is possible to implement an efficient version of such a converter and that it can reduce the end-to-end application delay in lossy networks.

In Chapter 5, we evaluated MPTCP in a different environment where it can bring benefits. The literature already showed that the multiple paths of data center networks can be used by MPTCP to improve performance. In this chapter, we evaluated whether the current design of data centers is a good fit for MPTCP. We learned that the current load balancing technique brings a limitation for multipath protocols as it is difficult to ensure that subflows follow disjoint paths in the network. We proposed a new way of performing load balancing that allows a source to ensure that a flow will follow a specific path without requiring a modification to the format of the packets. MPTCP can benefit from it by establishing as many subflows as there exist paths between a source and a destination in order to fully utilize the network. We showed that this new load-balancing technique gives similar load balancing performance for uncontrolled flows and improve the overall

performances.

6.2 Perspectives and further work

In this thesis we focused on the deployability of MPTCP on the client side. To be fully deployed on both the client and the server side middleboxes, such as firewall, have also to be updated to support MPTCP. This may raise many challenges. For example, some implementations of server load balancing use a hash function on the 5-tuple to redirect incoming connections to servers. This allows to perform a stateless load balancing. With MPTCP subflows will have different hash results and so might be redirected to different servers. The same problems occurs when enabling IPv6 with Happy Eyeballs. This current stateless implementation of load balancer is therefore not enough in a MPTCP-enabled world. Two possible solutions are either to modify the protocol so that each packet contains a unique identifier that can be used to load-balance subflows or load balancers must terminate the MPTCP connections or maintain some state.

In this thesis we have proposed and evaluated different tools and mechanisms that bring benefit to MPTCP in specific situations. `tracebox` opens new directions to allow researchers to better understand the deployment of middleboxes in the global Internet. As of the writing of this thesis, `tracebox` is still a work in progress. The `tracebox` API can be improved to simplify the scripting capability as well as to provide better support for various layers such as SCTP, GRE, LISP, etc. The evaluation we have performed on `tracebox` was mainly to assess that `tracebox` can help understanding and detecting middleboxes. PlanetLab is not the right place to actually detect middleboxes, even if we detected some of them. Many other measurement platforms exist, such as RIPE Atlas [RIP13] or SamKnows [sam13]. These measurement platforms are a better place to perform middlebox detection, they are placed behind commercial providers and not university network providers as PlanetLab nodes. They allow to perform large-scale measurement campaigns to analyze in more details middlebox interferences in IPv4 and IPv6 networks. `tracebox` could also be extended to contain a database of middleboxes footprints such that when it detects a packet modification it could fingerprint the middlebox responsible of this modification similarly as what Nmap [Lyo09] does to identify hosts on the Internet.

The protocol converter presented in Chapter 4 allows to transform a MPTCP connection into a regular TCP one and vice versa. Other researchers have proposed to use transparent proxies within the network [ARBW12a, XGHZ12], i.e., to use MPTCP if both end hosts only supports TCP. Our proposed protocol converter could be used in similar "double proxies" sce-

narios. It has to be evaluated to understand the behavior of the converter in such scenarios and whether it brings the desired features.

Our evaluation of the mobility using MPTCP could be extended to consider different environments and applications. Now more and more user share their WiFi and some operators activate WiFi sharing solutions such as FON on their set-top boxes. It could be interesting to evaluate the mobility of a device that only has a single or multiple WiFi interfaces. Moving from one access point to another with periods during which no connectivity is available. Evaluating different types of applications in this scenario can help understanding if smartphones can disable 3G/LTE in a crowded area where there exist a large number of free Hotspots.

Acronyms

3DES Triple DES.

ACK Acknowledgement.

ADSL Asymmetric Digital Subscriber Line.

AES Advanced Encryption Standard.

ALG Application Level Gateway.

ALTO Application-Layer Traffic Optimization.

API Application Programming Interface.

ATM Asynchronous transfer mode.

BFD Bidirectional Forwarding Detection.

BGP Border Gateway Protocol.

CDF Cumulative Distribution Function.

CFLB Controllable per-Flow Load Balancing.

CPU Central Processing Unit.

CRC Cyclic Redundancy Check.

DACK Data Acknowledgement.

DAG Directed Acyclic Graph.

DES Data Encryption Standard.

DNS Domain Name System.

DSN Data Sequence Number.

DSS Data Sequence Signal.

ECMP Equal Cost Multi-Path.

ECN Explicit Congestion Notification.

FIN Finish.

FPE Format-Preserving Encryption.

FTP File Transfer Protocol.

GRE Generic Routing Encapsulation.

HMAC keyed-Hash Message Authentication Code.

HTTP HyperText Transfer Protocol.

IANA Internet Assigned Numbers Authority.

iBGP internal Border Gateway Protocol.

ICMP Internet Control Message Protocol.

IDS Intrusion Detection System.

IDSN Initial Data Sequence Number.

IETF Internet Engineering Task Force.

IGP Interior Gateway Protocol.

IP Internet Protocol.

IPv4 Internet Protocol version 4.

IPv6 Internet Protocol version 6.

IS-IS Intermediate System to Intermediate System.

ISN Initial Sequence Number.

ISP Internet Service Provider.

LDP Label Distribution Protocol.

LISP Locator/Identifier Separation Protocol.

LSP Label Switched Path.

LSR Label Switched Router.

LTE Long Term Evolution.

MD5 Message Digest 5.

MiMBox Multipath in the Middle(Box).

MIPv6 Mobile IPv6.

MPLS Multiprotocol Label Switching.

MPLS-TP MPLS Transport Profile.

MPTCP Multipath TCP.

MSS Maximum Segment Size.

MTU Maximum Transmission Unit.

NAT Network Address Translator.

NIC Network Interface Controller.

NOP No Operation.

OS Operating System.

OSI Open Systems Interconnection.

OSPF Open Shortest Path First.

PE Provider Edge.

RBridges Routing Bridges.

RFC Request For Comments.

RFS Receive Flow Steering.

RST Reset.

RSVP-TE Resource Reservation Protocol – Traffic Engineering.

RTO Retransmission Timeout.

RTP Real-time Transport Protocol.

RTT Round-Trip Time.

SACK Selective Acknowledgement.

SCTP Stream Control Transmission Protocol.

SCTP-CMT SCTP Concurrent Multipath Transfer.

SLA Service Level Agreement.

SOCKS Socket Secure.

SYN Synchronize.

TCP Transmission Control Protocol.

TRILL Transparent Interconnection of Lots of Links.

TTL Time To Live.

UDP User Datagram Protocol.

VLAN Virtual LAN.

VoIP Voice over IP.

VP Vantage Point.

VPN Virtual Private Network.

WCCP Web Cache Communication Protocol.

WiFi Wireless Fidelity.

XCB eXtended CodeBook.

Bibliography

[3GP] 3GPP. 3GPP TS 36.331 – Radio Resource Control (RRC). Technical report. See `http://www.3gpp.org/ftp/Specs/html-info/36331.htm`.

[3GP12] 3GPP. 3GPP TS 36.412 V11.0.0 – Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Access Network (E-UTRAN); S1 signaling transport (Release 8). Technical report, 2012. See `http://www.3gpp.org/ftp/Specs/html-info/36412.htm`.

[Abb99] J. Abbate. *Inventing the Internet*. MIT Press, 1999.

[ABG+01] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow. RSVP-TE: Extensions to RSVP for LSP Tunnels. RFC 3209 (Proposed Standard), December 2001.

[abi09] ABI Research Forecasts Wi-Fi Access Point Shipments to Exceed 70 Million by 2010. Technical report, ABI Research, 2009. See `hhttp://www.businesswire.com/news/home/20090310005809/en/ABI-Research-Forecasts-Wi-Fi-Access-Point-Shipments`.

[ACGP08] G. Anastasi, M. Conti, E. Gregori, and A. Passarella. 802.11 power-saving mode for mobile computing in wi-fi hotspots: Limitations, enhancements and open issues. *Wirel. Netw.*, 14(6):745–768, December 2008.

[ACO+06] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding traceroute anomalies with Paris traceroute. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 153–158, New York, NY, USA, 2006. ACM.

[AFLV08] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communica-*

tion, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[AFRR+10] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 19–19, Berkeley, CA, USA, 2010. USENIX Association.

[AFT07] B. Augustin, T. Friedman, and R. Teixeira. Measuring load-balanced paths in the internet. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, IMC '07, pages 149–160, New York, NY, USA, 2007. ACM.

[AL13] Alcated-Lucent. MPTCP Proxy, October 2013. See `https://open-innovation.alcatel-lucent.com/projects/mptcp-proxy`.

[AMT07] L. Andersson, I. Minei, and B. Thomas. LDP Specification. RFC 5036 (Draft Standard), October 2007. Updated by RFCs 6720, 6790.

[Apa13] Apache. Traffic server, 2013. `http://trafficserver.apache.org/`.

[app14] iOS: Multipath TCP Support in iOS 7, January 2014. See `http://support.apple.com/kb/HT5977`.

[APY13] R. Alimi, R. Penno, and Y. Yang. ALTO Protocol. Internet-Draft draft-ietf-alto-protocol-18, 2013.

[ARBW12a] T. Ayar, B. Rathke, L. Budzisz, and A. Wolisz. A Transparent Performance Enhancing Proxy Architecture To Enable TCP over Multiple Paths for Single-Homed Hosts. Internet-Draft draft-ayar-transparent-sca-proxy-00, February 2012.

[ARBW12b] T. Ayar, B. Rathke, L. Budzisz, and A. Wolisz. TCP over multiple paths revisited: Towards transparent proxy solutions. In *Communications (ICC), 2012 IEEE International Conference on*, pages 109–114, 2012.

[Ass08] I. S. Association. IEEE Std 802.1AX-2008 IEEE Standard for Local and Metropolitan Area Networks - Link Aggregation. `http://standards.ieee.org/getieee802/download/802.1AX-2008.pdf`, 2008.

[AVRG+13] A. Aucinas, N. Vallina-Rodriguez, Y. Grunenberger, V. Erramilli, K. Papagiannaki, J. Crowcroft, and D. Wetherall.

Staying online while mobile: The hidden costs. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 315–320, New York, NY, USA, 2013. ACM.

[BAAZ10] T. Benson, A. Anand, A. Akella, and M. Zhang. The case for fine-grained traffic engineering in data centers. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, INM/WREN'10, pages 2–2, Berkeley, CA, USA, 2010. USENIX Association.

[Bak95] F. Baker. Requirements for IP Version 4 Routers. RFC 1812 (Proposed Standard), June 1995.

[BAM10] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[BB95] A. Bakre and B. R. Badrinath. I-TCP: indirect TCP for mobile hosts. In *Distributed Computing Systems, 1995., Proceedings of the 15th International Conference on*, pages 136–143, 1995.

[BBX10] R. Beverly, A. Berger, and G. G. Xie. Primitives for active internet topology mapping: toward high-frequency characterization. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, IMC '10, pages 165–171, New York, NY, USA, 2010. ACM.

[Ber96] D. Bernstein. SYN cookies, 1996. See `http://cr.yp.to/syncookies.html`.

[BF13] T. Bourgeau and T. Friedman. Efficient IP-Level network topology capture. In *Proceedings of the 14th international conference on Passive and Active Measurement*, PAM'13, pages 11–20, Berlin, Heidelberg, 2013. Springer-Verlag.

[BFD+11] S. Bryant, C. Filsfils, U. Drafz, V. Kompella, J. Regan, and S. Amante. Flow-Aware Transport of Pseudowires over an MPLS Packet Switched Network. RFC 6391 (Proposed Standard), November 2011.

[BGTP07] R. Bonica, D. Gan, D. Tappan, and C. Pignataro. Extended ICMP to Support Multi-Part Messages. RFC 4884 (Proposed Standard), April 2007.

[Bio] P. Biondi. Scapy. See `http://www.secdev.org/projects/scapy/`.

[BKG+01] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135 (Informational), June 2001.

[BMV10] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting mobile 3G using WiFi. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 209–222, New York, NY, USA, 2010. ACM.

[Bon13] O. Bonaventure. Apple seems to also believe in Multipath TCP, 2013. See `http://perso.uclouvain.be/olivier.bonaventure/blog/html/2013/09/18/mptcp.html`.

[BPB11] S. Barré, C. Paasch, and O. Bonaventure. MultiPath TCP: from theory to practice. In *Proceedings of the 10th international IFIP TC 6 conference on Networking - Volume Part I*, NETWORKING'11, pages 444–457, Berlin, Heidelberg, 2011. Springer-Verlag.

[BPSK97] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz. A comparison of mechanisms for improving TCP performance over wireless links. *IEEE/ACM Trans. Netw.*, 5(6):756–769, December 1997.

[BRB11] S. Barré, J. Ronan, and O. Bonaventure. Implementation and evaluation of the shim6 protocol in the linux kernel. *Computer Communications*, 34(14):1685–1695, 2011.

[Bri95] T. Brisco. DNS Support for Load Balancing. RFC 1794 (Informational), April 1995.

[CA11] B. Carpenter and S. Amante. Using the IPv6 Flow Label for Equal Cost Multipath Routing and Link Aggregation in Tunnels. RFC 6438 (Proposed Standard), November 2011.

[Cas01] M. Castells. *The Internet Galaxy : Reflections on the Internet, Business, and Society*. Oxford University Press, reprint edition, April 2001.

[CB02] B. Carpenter and S. Brim. Middleboxes: Taxonomy and Issues. RFC 3234 (Informational), February 2002.

[CDG06] A. Conta, S. Deering, and M. Gupta. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 4443 (Draft Standard), March 2006. Updated by RFC 4884.

[CDK09] C. Cannière, O. Dunkelman, and M. Knežević. KATAN and KTANTAN – A Family of Small and Efficient Hardware-Oriented Block Ciphers. In *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '09, pages 272–288, Berlin, Heidelberg, 2009. Springer-Verlag.

[CGM13] S. Cheshire, J. Graessley, and R. McGuire. Encapsulation of TCP and other Transport Protocols over UDP. Internet-Draft draft-cheshire-tcp-over-udp-00, June 2013.

[Cis98] Load balancing with Cisco Express Forwarding. Technical report, Cisco Systems, Inc., 1998. See `http://www.cisco.com/en/US/products/hw/modules/ps2033/prod_technical_reference09186a00800afeb7.html`.

[CKY11] A. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *INFOCOM, 2011 Proceedings IEEE*, pages 1629–1637, 2011.

[Cla88] D. Clark. The design philosophy of the DARPA internet protocols. In *Symposium proceedings on Communications architectures and protocols*, SIGCOMM '88, pages 106–114, New York, NY, USA, 1988. ACM.

[CMT+11] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, pages 254–265, New York, NY, USA, 2011. ACM.

[Cox96] A. Cox. Kernel Korner: Network Buffers and Memory Management. *Linux J.*, 1996(30es), October 1996.

[CS05] I. Cisco Systems. Load Splitting IP Multicast Traffic over ECMP, May 2005. See `http://www.cisco.com/en/US/docs/ios/ipmulti/configuration/guide/imc_load_splt_ecmp.pdf`.

[CS13a] I. Cisco System. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2012-2017, February 2013. See `http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.html`.

[CS13b] I. Cisco Systems. Cisco nexus 5000 series switches, 2013. See `http://www.cisco.com/en/US/products/ps9670/`.

[CS13c] I. Cisco Systems. IOS IP Service Level Agreements (SLAs), 2013. See `http://www.cisco.com/en/US/products/ps6602/products_ios_protocol_group_home.html`.

[CS13d] I. Cisco Systems. Server Cluster Designs with Ethernet, 2013. See `http://www.cisco.com/en/US/docs/solutions/Enterprise/Data_Center/DC_Infra2_5/DCInfra_3.html`.

[CWZ00] Z. Cao, Z. Wang, and E. Zegura. Performance of hashing-based schemes for Internet load balancing. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 332–341 vol.1, 2000.

[DCVS13] L. D'Acunto, N. Chiluka, T. Vinkó, and H. Sips. BitTorrent-like P2P approaches for VoD: A comparative study. *Comput. Netw.*, 57(5):1253–1276, April 2013.

[Det13] G. Detal. `tracebox`, July 2013. See `http://www.tracebox.org`.

[DF07] B. Donnet and T. Friedman. Internet topology discovery: a survey. *IEEE Communications Surveys and Tutorials*, 9(4), December 2007.

[DRFC05] B. Donnet, P. Raoult, T. Friedman, and M. Crovella. Efficient algorithms for large-scale topology discovery. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '05, pages 327–338, New York, NY, USA, 2005. ACM.

[dVPC+08] G. V. de Velde, C. Popoviciu, T. Chown, O. Bonness, and C. Hahn. IPv6 Unicast Address Assignment Considerations. RFC 5375 (Informational), December 2008.

[Edd04] W. M. Eddy. At what layer does mobility belong? *Comm. Mag.*, 42(10):155–159, October 2004.

[ETS12] ETSI. Network Functions Virtualisation – An Introduction, Benefits, Enablers, Challenges & Call for Action. Technical report, oct 2012. See `http://portal.etsi.org/NFV/NFV_White_Paper.pdf`.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[fon13] Fon, October 2013. See `http://www.fon.com`.

[FRHB13] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824 (Experimental), January 2013.

[GGSS09] P. B. Godfrey, I. Ganichev, S. Shenker, and I. Stoica. Pathlet routing. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 111–122, New York, NY, USA, 2009. ACM.

[GHJ+09] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 51–62, New York, NY, USA, 2009. ACM.

[GLL+09] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: a high performance, server-centric network architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 63–74, New York, NY, USA, 2009. ACM.

[Gro06] I. Groenbaek. Conversion Between the TCP and ISO Transport Protocols as a Method of Achieving Interoperability Between Data Communications Systems. *IEEE J.Sel. A. Commun.*, 4(2):288–296, September 2006.

[hap12] HAProxy – The Reliable, High Performance TCP/HTTP Load Balancer, October 2012. See `http://haproxy.1wt.eu`.

[HDP+13] B. Hesmans, F. Duchene, C. Paasch, G. Detal, and O. Bonaventure. Are tcp extensions middlebox-proof? In *CoNEXT workshop HotMiddlebox*. ACM, December 2013.

[Hef98] A. Heffernan. Protection of BGP Sessions via the TCP MD5 Signature Option. RFC 2385 (Proposed Standard), August 1998.

[Her10] T. Herbert. RFS: Receive Flow Steering, 2010. See `http://lwn.net/Articles/381955/`.

[Hes13] B. Hesmans. Mbclick, July 2013. See `https://bitbucket.org/bhesmans/mbclick`.

[HH99] B. Harris and R. Hunt. TCP/IP security threats and attack methods. *Computer Communications*, 22(10):885 – 897, 1999.

[HJPM10] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[HK12] G. Hampel and T. Klein. MPTCP Proxies and Anchors. Internet-Draft draft-hampel-mptcp-proxies-anchors-00, February 2012.

[HNR+11] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, IMC '11, pages 181–194, New York, NY, USA, 2011. ACM.

[Hop00] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992 (Informational), November 2000.

[HRI+08] H. Haddadi, M. Rio, G. Iannaccone, A. Moore, and R. Mortier. Network topologies: inference, modeling, and generation. *Communications Surveys Tutorials, IEEE*, 10(2):48–69, 2008.

[HV04] A. Hodjat and I. Verbauwhede. A 21.54 Gbits/s Fully Pipelined AES Processor on FPGA. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '04, pages 308–309, Washington, DC, USA, 2004. IEEE Computer Society.

[IAS06] J. R. Iyengar, P. D. Amer, and R. Stewart. Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths. *IEEE/ACM Trans. Netw.*, 14(5):951–964, October 2006.

[ICBD04] G. Iannaccone, C.-N. Chuah, S. Bhattacharyya, and C. Diot. Feasibility of IP restoration in a tier 1 backbone. *Netwrk. Mag. of Global Internetwkg.*, 18(2):13–19, March 2004.

[IdFF96] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho. Lua - an extensible extension language. *Softw. Pract. Exper.*, 26(6):635–652, June 1996.

[int13] INTERNET USAGE STATISTICS – The Internet Big Picture, October 2013. See `http://www.internetworldstats.com/stats.htm`.

[ipe13] Iperf, July 2013. See `http://iperf.sourceforge.net/`.

[Jac88] V. Jacobson. Congestion avoidance and control. In *Symposium proceedings on Communications architectures and pro-*

tocols, SIGCOMM '88, pages 314–329, New York, NY, USA, 1988. ACM.

[JBB92] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323 (Proposed Standard), May 1992.

[JNW10] B. Jacob, S. Ng, and D. Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[Jon13] R. Jones. Netperf, Sept 2013. See `http://www.netperf.org/netperf/`.

[KBMA+10] E. Katz-Bassett, H. V. Madhyastha, V. K. Adhikari, C. Scott, J. Sherry, P. Van Wesep, T. Anderson, and A. Krishnamurthy. Reverse traceroute. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 15–15, Berkeley, CA, USA, 2010. USENIX Association.

[KCL04] S. J. Koh, M. J. Chang, and M. Lee. mSCTP for soft handover in transport layer. *Communications Letters, IEEE*, 8(3):189–191, 2004.

[KDA+12] K. Kompella, J. Drake, S. Amante, W. Henderickx, and L. Yong. The Use of Entropy Labels in MPLS Forwarding. RFC 6790 (Proposed Standard), November 2012.

[KGP+12] R. Khalili, N. Gast, M. Popovic, U. Upadhyay, and J.-Y. Le Boudec. MPTCP is not pareto-optimal: performance issues and a possible solution. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, pages 1–12, New York, NY, USA, 2012. ACM.

[KH13] K. Kompella and M. Hellers. Multi-path Label Switched Paths Signaled Using RSVP-TE. Internet-Draft draft-kompella-mpls-rsvp-ecmp-04, 2013.

[KKG+10] M. E. Kounavis, X. Kang, K. Grewal, M. Eszenyi, S. Gueron, and D. Durham. Encrypting the internet. In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 135–146, New York, NY, USA, 2010. ACM.

[KKL+07] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *Ottawa Linux Symposium*, pages 225–230, July 2007.

[KKSB07] S. Kandula, D. Katabi, S. Sinha, and A. Berger. Dynamic load balancing without packet reordering. *SIGCOMM Comput. Commun. Rev.*, 37(2):51–62, March 2007.

[KKW+03] H. T. Kaur, S. Kalyanaraman, A. Weiss, S. Kanwar, and A. Gandhi. BANANAS: an evolutionary framework for explicit and multipath routing in the internet. In *Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, FDNA '03, pages 277–288, New York, NY, USA, 2003. ACM.

[KMC+00] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000.

[KS06] K. Kompella and G. Swallow. Detecting Multi-Protocol Label Switched (MPLS) Data Plane Failures. RFC 4379 (Proposed Standard), February 2006. Updated by RFCs 5462, 6424, 6425, 6426, 6829.

[KSG+09] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, IMC '09, pages 202–208, New York, NY, USA, 2009. ACM.

[KW10] D. Katz and D. Ward. Bidirectional Forwarding Detection (BFD). RFC 5880 (Proposed Standard), June 2010.

[LHH08] M. Luckie, Y. Hyun, and B. Huffaker. Traceroute probe method and forward IP path inference. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, IMC '08, pages 311–324, New York, NY, USA, 2008. ACM.

[Lyo09] G. F. Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning.* Insecure, USA, 2009.

[MAB+08] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[MAF04] A. Medina, M. Allman, and S. Floyd. Measuring interactions between transport protocols and middleboxes. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, IMC '04, pages 336–341, New York, NY, USA, 2004. ACM.

[MB00] D. A. Maltz and P. Bhagwat. TCP splice application layer proxy performance. *J. High Speed Netw.*, 8(3):225–240, January 2000.

[MC09] A. Morton and B. Claise. Packet Delay Variation Applicability Statement. RFC 5481 (Informational), March 2009.

[McL12] D. McLaggan. Web Cache Communication Protocol V2, Revision 1. Internet-Draft draft-mclaggan-wccp-v2rev1-00, August 2012.

[MD90] J. Mogul and S. Deering. Path MTU discovery. RFC 1191 (Draft Standard), November 1990.

[MEFV08] M. Motiwala, M. Elmore, N. Feamster, and S. Vempala. Path splicing. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 27–38, New York, NY, USA, 2008. ACM.

[MF04] D. A. McGrew and S. R. Fluhrer. The Extended Codebook (XCB) Mode of Operation. Cryptology ePrint Archive, Report 2004/278, 2004. See `http://eprint.iacr.org/`.

[MFB+11] P. Mérindol, P. Francois, O. Bonaventure, S. Cateloin, and J. J. Pansiot. An efficient algorithm to enable path diversity in link state routing networks. *Comput. Netw.*, 55(5):1132–1149, April 2011.

[Mic99] Microsoft. Patch available to improve TCP initial sequence number randomness. Microsoft Security Bulletin MS99-066, Microsoft, October 1999. See `http://technet.microsoft.com/en-us/security/bulletin/ms99-046`.

[MMFR96] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), October 1996.

[MMH07] R. Martin, M. Menth, and M. Hemmkeppler. Accuracy and Dynamics of Multi-Stage Load Balancing for Multipath Internet Routing. In *IEEE International Conference on Communications (ICC)*, Glasgow, Scotland, 6 2007.

[MN06] R. Moskowitz and P. Nikander. Host Identity Protocol (HIP) Architecture. RFC 4423 (Informational), May 2006.

[MYAFM10] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. SPAIN: COTS data-center Ethernet for multipathing over arbitrary topologies. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*,

NSDI'10, pages 18–18, Berkeley, CA, USA, 2010. USENIX Association.

[NB09] E. Nordmark and M. Bagnulo. Shim6: Level 3 Multihoming Shim Protocol for IPv6. RFC 5533 (Proposed Standard), June 2009.

[PCVB13] C. Pelsser, L. Cittadini, S. Vissicchio, and R. Bush. From paris to tokyo: On the suitability of ping to measure latency. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 427–432, New York, NY, USA, 2013. ACM.

[Pel13] E. Pellegrino. libcrafter – a high level library for c++ to generate and sniff network packets, July 2013. See `https://code.google.com/p/libcrafter/`.

[Per04] R. Perlman. Rbridges: transparent routing. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 1211–1218 vol.2, 2004.

[Per10] C. Perkins. IP Mobility Support for IPv4, Revised. RFC 5944 (Proposed Standard), November 2010.

[per12] perf: Linux profiling with performance counters, October 2012. See `https://perf.wiki.kernel.org`.

[PJA11] C. Perkins, D. Johnson, and J. Arkko. Mobility Support in IPv6. RFC 6275 (Proposed Standard), July 2011.

[Pos81] J. Postel. Internet Control Message Protocol. RFC 792 (INTERNET STANDARD), September 1981. Updated by RFCs 950, 4884, 6633.

[PSZM12] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In P. Felber, F. Bellosa, and H. Bos, editors, *EuroSys*, pages 337–350. ACM, 2012.

[PWH+10] A. Pathak, Y. A. Wang, C. Huang, A. Greenberg, Y. C. Hu, R. Kern, J. Li, and K. W. Ross. Measuring and evaluating TCP splitting for cloud services. In *Proceedings of the 11th international conference on Passive and active measurement*, PAM'10, pages 41–50, Berlin, Heidelberg, 2010. Springer-Verlag.

[QWG+11] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Profiling resource usage for mobile applica-

tions: A cross-layer approach. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 321–334, New York, NY, USA, 2011. ACM.

[RBP+11] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath TCP. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, pages 266–277, New York, NY, USA, 2011. ACM.

[RCCD04] J. Rajahalme, A. Conta, B. Carpenter, and S. Deering. IPv6 Flow Label Specification. RFC 3697 (Proposed Standard), March 2004.

[RFB01] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), September 2001. Updated by RFCs 4301, 6040.

[RHW11] C. Raiciu, M. Handley, and D. Wischik. Coupled Congestion Control for Multipath Transport Protocols. RFC 6356 (Experimental), October 2011.

[RIP13] RIPE, NCC. RIPE Atlas, 2013. See `https://atlas.ripe.net/`.

[Riv95] R. Rivest. The RC5 encryption algorithm. In B. Preneel, editor, *Fast Software Encryption*, volume 1008 of *Lecture Notes in Computer Science*, pages 86–96. Springer Berlin Heidelberg, 1995.

[Riz12] L. Rizzo. Netmap: a novel framework for fast packet i/o. In *Proc. of the USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9. USENIX Association, 2012.

[RNBH11] C. Raiciu, D. Niculescu, M. Bagnulo, and M. J. Handley. Opportunistic mobility with multipath TCP. In *Proceedings of the sixth international workshop on MobiArch*, MobiArch '11, pages 7–12, New York, NY, USA, 2011. ACM.

[RPB+12] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? designing and implementing a deployable multipath TCP. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[RR02] M.-C. Roşu and D. Roşu. An evaluation of TCP splice benefits in web proxy servers. In *Proceedings of the 11th international conference on World Wide Web*, WWW '02, pages 13–24, New York, NY, USA, 2002. ACM.

[Rus12] K. Rushton. Number of smartphones tops one billion, October 2012. See `http://www.telegraph.co.uk/finance/9616011/Number-of-smartphones-tops-one-billion.html`.

[RVC01] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. RFC 3031 (Proposed Standard), January 2001.

[SACA] C. Shannon, E. Aben, K. Claffy, and D. Andersen. The CAIDA Anonymized 2008 Internet Traces – 2008-07-17 12:59:07 - 2008-07-17 14:01:00. `http://www.caida.org/data/passive/passive_2008_dataset.xml`.

[sam13] SamKnows – Accurate broadband performance information for consumers, governments and ISPs, 2013. See `http://www.samknows.com/`.

[San13] Sandvine. Global Internet Phenomena Report 1H 2013. Technical report, 2013. See `https://www.sandvine.com/downloads/general/global-internet-phenomena/2013/sandvine-global-internet-phenomena-report-1h-2013.pdf`.

[Sav06] P. Savola. MTU and Fragmentation Issues with In-the-Network Tunneling. RFC 4459 (Informational), April 2006.

[SB00] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, MobiCom '00, pages 155–166, New York, NY, USA, 2000. ACM.

[SH99] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663 (Informational), August 1999.

[SHHP00] O. Spatscheck, J. S. Hansen, J. H. Hartman, and L. L. Peterson. Optimizing TCP forwarder performance. *IEEE/ACM Trans. Netw.*, 8(2):146–157, April 2000.

[SHS+12] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: network processing as a cloud service. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications,*

technologies, architectures, and protocols for computer communication, SIGCOMM '12, pages 13–24, New York, NY, USA, 2012. ACM.

[SJGH10] S. Sen, Y. Jin, R. Guérin, and K. Hosanagar. Modeling the dynamics of network technology adoption and the role of converters. *IEEE/ACM Trans. Netw.*, 18(6):1793–1805, December 2010.

[SKT96] J. D. Salehi, J. F. Kurose, and D. Towsley. The effectiveness of affinity-based scheduling in multiprocessor network protocol processing (extended version). *IEEE/ACM Trans. Netw.*, 4(4):516–530, August 1996.

[SMWA04] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring ISP topologies with rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1):2–16, February 2004.

[squ12] Squid: Optimising Web Delivery, October 2012. See `http://www.squid-cache.org`.

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, November 1984.

[Ste07] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), September 2007.

[SXM+00] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960 (Proposed Standard), October 2000.

[SXT+07] R. Stewart, Q. Xie, M. Tuexen, S. Maruyama, and M. Kozuka. Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration. RFC 5061 (Proposed Standard), September 2007.

[Tor01] M. Torren. tcptraceroute - a traceroute implementation using TCP packets. man page, UNIX, 2001. See source code: `http://michael.toren.net/code/tcptraceroute/`.

[TP09] J. Touch and R. Perlman. Transparent Interconnection of Lots of Links (TRILL): Problem and Applicability Statement. RFC 5556 (Informational), May 2009.

[V. 89] V. Jacobson et al. traceroute. man page, UNIX, 1989. See source code: `ftp://ftp.ee.lbl.gov/traceroute.tar.gz`.

[VF07] B. Veal and A. Foong. Performance scalability of a multi-core web server. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, ANCS '07, pages 57–66, New York, NY, USA, 2007. ACM.

[Vos10] E. Vos. AT&T sets up free WiFi in Times Square NYC to offload 3G data traffic, May 2010. See `http://www.muniwireless.com/2010/05/25/att-free-wifi-offload-data-traffic-nyc/`.

[WDC11] W. Wu, P. DeMar, and M. Crawford. Why Can Some Advanced Ethernet NICs Cause Packet Reordering? *Communications Letters, IEEE*, 15(2):253–255, 2011.

[wei13] weighttp, October 2013. See `http://redmine.lighttpd.net/projects/weighttp/wiki`.

[WHB08] D. Wischik, M. Handley, and M. B. Braun. The resource pooling principle. *SIGCOMM Comput. Commun. Rev.*, 38(5):47–52, September 2008.

[WLT+11] H. Warma, T. Leva, H. Tripp, A. Ford, and A. Kostopoulos. Dynamics of communication protocol diffusion: the case of multipath TCP. *NETNOMICS: Economic Research and Electronic Networking*, 12(2):133–159, 2011.

[WQX+11] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, pages 374–385, New York, NY, USA, 2011. ACM.

[WT86] A. F. Webster and S. E. Tavares. On the Design of S-Boxes. In *Advances in Cryptology*, CRYPTO '85, pages 523–534, London, UK, UK, 1986. Springer-Verlag.

[XGHZ12] K. Xue, J. Guo, P. Hong, and L. Zhu. TMPP for Both Two MPTCP-unaware Hosts. Internet-Draft draft-xue-mptcp-tmpp-unware-hosts-00, 2012.

[XLC11] K. Xi, Y. Liu, and J. Chao. Enabling Flow-based Routing Control in Data Center Networks using Probe and ECMP. In *INFOCOM Workshop on cloud computing*, pages 614–619, 2011.

[YW06] X. Yang and D. Wetherall. Source selectable path diversity via routing deflections. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, pages 159–170, New York, NY, USA, 2006. ACM.