Université Catholique de Louvain

École Polytechnique de Louvain

Multipath TCP with real Smartphone applications

Advisor: Pr. Olivier Bonaventure
Readers: Patrick Delcoigne
Benjamin Hesmans
Pr. Ramin Sadre

Authors:
Matthieu Baerts
Quentin De Coninck

Thesis submitted for the Master's degrees in Computer Science and Computer Engineering

Academic Year 2014-2015 — June session

There are many people we would like to thank.

Our advisor of course, Professor Bonaventure, for his patience, availability and great help to push us moving forward.

Benjamin Hesmans, Viet-Hoang Tran, Fabien Duchêne and Gregory Detal for their kind support all along this year.

Gregory Detal and Sébastien Barré for their huge work in backporting latest version of Multipath TCP for the Nexus 5.

Proximus, and in particular Patrick Delcoigne and his team, for the Nexus 5 devices and the cellular network to perform our measurements.

We also would like to thanks our families and friends for their precious support and advices.

Also, special thanks to our friends at Clinch team, to keep continuously an excellent mood in Parnas room.

Contents

Abstract

Multipath TCP is a new TCP extension that allows spreading data across several paths. Smartphone are multi-homed devices that can take advantage of such ability. Compared to regular TCP, Multipath TCP is able to keep connections alive across different networks.

This work explores three measurement campaigns. The first one analyses how Multipath TCP reacts on smartphones relative to TCP with predefined scenarios on real applications and partial control of the environment. We propose and implement an automated measurement framework and observe that Multipath TCP works on smartphones without requiring any change on the application. With the default scheduler, quite long connections are balanced on both interfaces depending of their performance, but behaviour of short connections mainly depends on the initial interface. The second one extends our analysis to real users and compare the traffic with the one seen by a Multipath TCP server. We show the heterogeneity of the subflows and observe that most of the subflows created by smartphones are not used. Multipath TCP's reinjections are quite rare, but can happen in burst, particularly in mobility scenarios. The third one studies the performance of Multipath TCP with the streaming use case. We notice that Multipath TCP offers a better service than TCP to its handover mechanisms. However, the current behaviour of Multipath TCP is not always what smartphone users expect and we propose several possible improvements for the current implementation.

Chapter 1

Introduction

In the past decade, smartphones and tablets became the most popular mobile devices. They contribute to a growing fraction of the Internet traffic, and this growth is planned to be exponential [14]. This kind of devices is usually able to be connected via WiFi and cellular networks. Then, the end-users expect to have their mobile devices capable to benefit seamlessly from all available interfaces. However, this coexistence between WiFi and cellular is much more complex and suffers from early design decisions of popular protocols.

In particular, the Transmission Control Protocol (TCP) is the dominant transport protocol in today's networks and is used by most applications running on smartphones. Though, its design is from the 1970s, and doesn't take into account the multihomed hosts like smartphones.

To address this last point, several technologies have been proposed during the last years [43] and some have be deployed. Multipath TCP is one of them and seems to be the more promising one. Indeed, this solution received a lot of attention when it was selected by Apple to support its voice recognition (Siri) application. Siri leverages Multipath TCP to send voice samples over both WiFi and cellular interfaces to cope with various failure scenarios. To our knowledge, Siri is the only widely deployed smartphone application that uses Multipath TCP. Despite this large deployed base, there is no public information about the performance benefits of using Multipath TCP with Siri.

Through this master thesis, we fill this gap by providing various measurements and analyses in different environments with smartphones using the Android operating system, the most deployed one on mobile devices[1]. After presenting a background about Multipath TCP in

[1] See http://www.businessinsider.com/iphone-v-android-market-share-2014-5?IR=T

chapter 2 and presenting required infrastructure in chapter 3, the next three chapters will be dedicated to our results. The first one will describe the behaviour of Multipath TCP with predefined application scenarios. A Multipath TCP capable SOCKS proxy is installed and the network traffic between the smartphone and the proxy will be examined. In particular, we analyse how is it balanced between the WiFi and the cellular interfaces and observe the situation from both sides, i.e. the smartphone and the proxy. The second one will extend our analyses to the traffic generated by a dozen of real users. Throughout this chapter, it will be compared with the traffic seen by a deployed Multipath TCP capable server. The overhead of Multipath TCP will also be studied. The third one will focus on the streaming use case. In particular, the benefits of Multipath TCP in mobility scenarios are observed, such as the support of handovers. Finally, the chapter 7 will concludes this thesis and will propose possible evolutions.

Chapter 2

Multipath TCP

Before going in details in our measurements, let us have some background information about Multipath TCP. More details can be found in [29, 51, 53]. This section assumes a basic understanding of TCP's principles. For more information about that, one can read [26].

2.1 Overview

Multipath TCP is an effort towards enabling the simultaneous use of several IP-addresses and interfaces. This is implemented thanks to a modification of TCP that presents a regular TCP [60] socket interface to applications while spreading data across several subflows. As stated in [63], the main design goals are the following ones:

- It must work with applications using regular TCP without requiring any changes to them.
- It must work in all scenarios where regular TCP currently works.
- It must be able to use the network at least as well as regular TCP, but without starving TCP.
- It must be implementable in operating systems without using excessive memory or processing.

Since Multipath TCP must be as usable as regular TCP for existing applications, it has to provide a reliable and in-order byte transmission service. This need of reliability comes with an acknowledgement mechanism. Although Multipath TCP allows to achieve a more efficient

use of the network, it can not overwhelm the receiver. To do so, it also provides a way for the receiver to perform flow control on the connection.

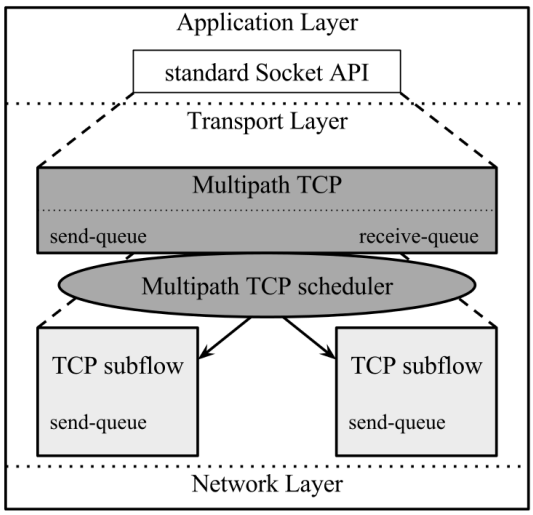


Figure 2.1: Architecture of Multipath TCP (Source: [51])

The architecture of the Multipath TCP implementation in the Linux kernel is shown on the figure 2.1. To the application, a standard stream socket interface is presented. At the transport layer, Multipath TCP is negotiated via new TCP options in the `SYN` packets of the TCP connections. To allow data transmissions across different paths, a *TCP subflow* is created along each of these paths. They look like regular TCP connections on the wire, including a 3-way handshake for the setup, a proper sequence number space with retransmissions and a 4-way handshake for their termination. These subflows are linked together to form the Multipath TCP connection and are used to carry the data between the end hosts. Note that those subflows will be also called *Multipath TCP subflows* later on in this thesis.

The *Multipath TCP scheduler* [55] is in charge of the distribition of the data across the different subflows — allowing to pool the resources of each subflow's path. This algorithm is used every time that a data segment needs to be sent. It selects, among the active subflows that have an open congestion window, the subflow that will be used to send the data. The default scheduler tries to send data over the subflow having the lowest round-trip-time. Note that another packet scheduler that selects the subflows in a round-robin fashion also exists [52]. Both of them allows the user to use a path as backup as explained in section 2.2.4 on page 7.

Each subflow uses its own sequence number space to detect losses and drive retransmissions. Multipath TCP adds connection-level sequence numbers to allow reordering at the receiver side. Finally, connection-level acknowledgements are used to implement proper flow controls.

In the remaining sections, we provide additional details about both parts of Multipath TCP: the control plane — responsible to create and destroy subflows and to signal other connection-level control information —, and the data plane — transmitting the data between the end hosts.

2.2 Control plane

This section describes the control information used by Multipath TCP. Those are sent within the TCP option space. Indeed, Multipath TCP uses a single TCP option type (30) and differentiates the control information using subtypes.

2.2.1 Initial handshake

The TCP three-way handshake is used to segment states between the client and the server. In particular, initial sequence numbers are exchanged and acknowledged and TCP options carried in the `SYN` and `SYN/ACK` are used to negotiate optional functionalities. During Multipath TCP's handshake, the end hosts detect whether the peer actually supports Multipath TCP or not. Three state variables of the connection are also exchanged during this initial handshake: the *token* to identify the Multipath TCP connection subflows belong to, the *Initial Data Sequence Number* to define the position of a segment inside the data stream, and two *keys* used as shared secret to authenticate the end hosts.

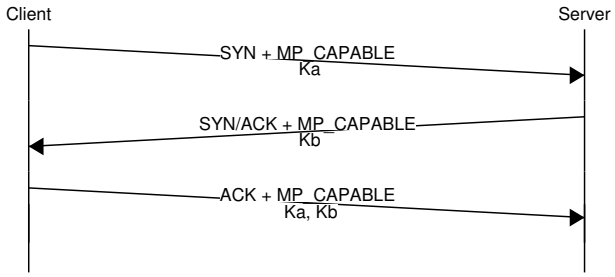


Figure 2.2: Handshake of the initial Multipath TCP subflow

Multipath TCP starts the connection by establishing an initial TCP subflow with a standard TCP 3-way handshake as shown on figure 2.2. It uses TCP options to specify that Multipath TCP is supported by the peer and to negotiate the above described three elements. Such detection of the Multipath TCP support is done by adding the `MP_CAPABLE` option inside the `SYN` packets. Exchanging the keys is achieved by sending them in clear inside the `MP_CAPABLE` option. Keys are echoed back in the third `ACK` of the 3-way handshake to let the servers handle stateless TCP handshakes [24]. To cope with middleboxes that could remove the `MP_CAPABLE` option from the `SYN+ACK` segment, an `MP_CAPABLE` option containing the two keys is also

placed in the third acknowledgement returned by the client. If one MP_CAPABLE option is missing in the three first messages, the connection falls back to TCP.

2.2.2 Handshake of the additional subflows

When adding a new subflow to a Multipath TCP connection, two problems must be solved. First, the new subflow needs to be associated with an existing Multipath TCP connection. The classical 5-tuple[1] cannot be used as a connection identifier, as it may change due to NATs. Secondly, Multipath TCP must be robust enough to avoid attackers to add his own subflow to an existing connection. Multipath TCP solves these two problems, first by using the locally unique token, and secondly by computing and verifying an HMAC, using the exchanged keys within the 3-way handshake of the initial subflow.

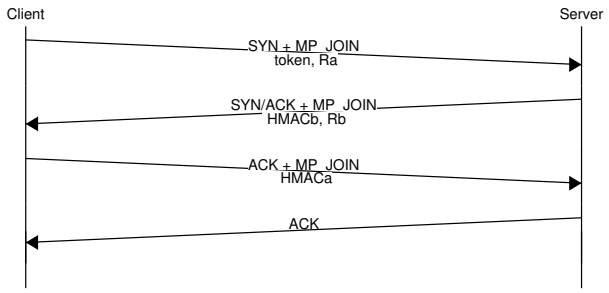


Figure 2.3: Handshake of additional Multipath TCP subflows

To open a new subflow, Multipath TCP performs a SYN exchange using the addresses and ports it wishes to use. The handshake is illustrated in figure 2.3. A TCP option, MP_JOIN, is added to the SYN. The token included inside the SYN allows to identify the Multipath TCP connection this subflow belongs to, while the 32 random bits (R_a) will be part of the input for the HMAC computation. The server computes the HMAC_b, based on two random numbers R_a and R_b and the keys exchanged in the initial handshake. Upon reception of the SYN/ACK, the client can verify the correctness of the HMAC_b, which proves that the server has knowledge of the keys K_a and K_b and thus participated in the initial handshake. Then, the client generates a different HMAC_a which is included inside the third ACK. This one allows the server to verify that the client is the same as the one involved in the initial handshake. Finally, the server sends a duplicate acknowledgement to the client, signaling the reception of this third ACK. The client will only start sending data to the server once it has received the fourth ACK. Once this is done, this subflow can be used to transmit data.

[1] The five elements are the following: protocol, IP_{src}, IP_{dst}, port_{src}, port_{dst}.

2.2.3 Removing a subflow

For each address is assigned an *address-ID*. This is an 8-bit integer that locally identifies the IP address. It is being used within the `MP_JOIN` option to let a host know which pair of address-IDs is associated to each subflow. The address-ID is mainly into the `REMOVE_ADDR` option to inform the peer when one of its addresses has become unavailable (e.g. the smartphone lost the connectivity to the WiFi access point). It specifies the address-ID that is associated to the IP address that should be removed. Upon reception of this option, the destination closes all TCP subflows that are using this address. The address-ID allows the host to identify which subflows to close, even if these ones have their public IP address changed due to a NAT. Such modification of the public IP address will not affect the address-ID and allows the ID to be used as an end-to-end identifier of the IP address.

2.2.4 Backup subflows

On smartphones, users might want to avoid sending traffic over the cellular interface to reduce the monetary cost of using the cellular network. Paasch et al. [54] added a mechanism inside Multipath TCP that allows to specify priorities on a subflow. The `MP_JOIN` option includes a backup bit that signals to the peer that it should not send any data on this subflow. This requirement can be violated by the peer if there is no other subflow available for data transmission. The `MP_PRIO` option allows to signal this kind of subflow priority after the 3-way handshake.

Specifying a backup bit on the cellular interface should reduce the traffic on that interface while still taking advantage of Multipath TCP's handover mechanism. Indeed, Multipath TCP opens TCP subflows over all interfaces, but the scheduler uses only a subset of these to transport data packets: non backup ones if available, else backup ones. Notice that Multipath TCP in backup mode recovers slower than in full mesh mode, because the congestion window on backup paths still have their initial value, whereas in full mesh mode, if paths were already used, their congestion window will be larger [54].

2.2.5 Path manager

The *path manager* defines the strategy that is used by the Multipath TCP stack to create subflows. Two path managers are included in the Linux kernel dedicated to Android smartphones: `full-mesh` and `ndiffports`. Their role is to decide when and how subflows are created. As of this writing, subflows are only created by the clients. The server does not attempt to create subflows as the clients are often behind NATs or firewalls that would block these subflows anyway [23].

The `full-mesh` path manager creates a subflow from each address owned by the client to each address advertised by the server. These subflows are created at the beginning of the connection, i.e. as soon as the initial subflow has been validated. If the client (or the server)

learns a new address, e.g. a smartphone attaches to a new WiFi access point, the `full-mesh` path manager automatically creates a new subflow over this interface.

The `ndiffports` path manager was designed for single-homed hosts in datacenters [61]. With this path manager, a Multipath TCP connection is composed of `n` subflows that use different source ports.

The `full-mesh` path manager is the default path manager in the Linux implementation.

2.3 Data plane

In this section, the data plane of Multipath TCP, responsible of the actual transmission of the byte stream sent and received by the application, is detailed.

2.3.1 A second sequence number space

A first dedicated sequence number space is required to provide a reliable and in-order delivery service to each TCP subflow. However, as Multipath TCP provides the same guarantees, it is necessary to allow the receiver to reorder the possibly out-of-order data segments. To illustrate this problem, consider the situation illustrated on figure 2.4. The smartphone wants to send four data packets to the server. Let us assume that the first packet is sent on the cellular interface (having a delay of 50 ms) and the three following on the WiFi interface (having a delay of 10 ms). The bandwidth is assumed to be large enough on both paths. In that case, the three last packets will arrive first on the server, while the first packet is still being transferred. With only sequence number space at the TCP subflows level, the server can manage reordering on a same path, but not between different ones. To cope with such cases, a second sequence number space at Multipath TCP connection level is added. It maps each byte to its position within the continuous data stream that is being sent by the application.

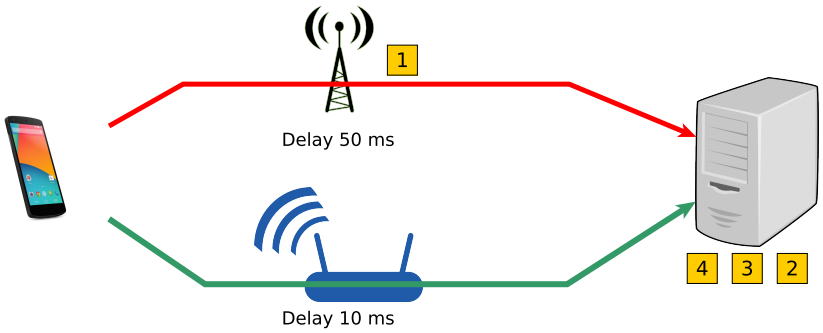


Figure 2.4: Example to show the need for a second sequence number space

The data sequence number space starts at the initial data sequence number. It is 64-bit long to prevent any issues with wrapped sequence numbers. With only 32 bits, if one path takes so much time to transfer a packet so that meanwhile another path could send 4 GBytes, the receiver will not be able to distinguish the packet with the old sequence number, from up-to-date data on the low-delay path. With 64 bits, the low-delay path must send 2^{64} bytes to have the same problem, which is unlikely to happen. This second sequence number, called *Data Sequence Number* (DSN), is contained in the *Data Sequence Signal* (DSS) option which is discussed in details in [29, 51]. The DSN corresponds to the bytestream. When data is sent over a subflow, its DSN is mapped to the regular sequence numbers with the DSS option. Note that the DSS option also contains DSN acknowledgements. Thanks to this option, the same data can be transmitted over more than one subflow. This is called a *reinjection* [63]. A reinjection can occur if data transmitted over one subflow has not been acknowledged quickly enough. For example, a smartphone could send some data over its WiFi interface while moving out of reach of the access point. This data will have to be retransmitted over the cellular interface to reach the server.

2.3.2 Congestion control

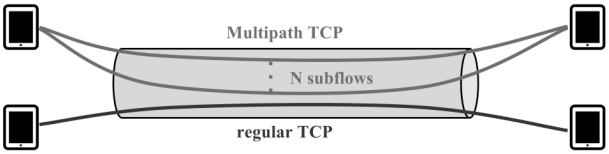


Figure 2.5: Multipath TCP with standard TCP congestion control (Source: [51])

Using several paths creates new problems from a congestion control point of view. With regular TCP, congestion occurs on one path between the client and the server. Multipath TCP uses several paths. Two paths will typically experience different levels of congestion. A naive solution to the congestion problem in Multipath TCP would be to use the standard TCP congestion control scheme on each subflow. This can be easily implemented but leads to unfairness with regular TCP. In the network illustrated in figure 2.5, two clients share the same bottleneck link. If the Multipath TCP-enabled client uses two subflows, then it will obtain two-thirds of the shared bottleneck. This is unfair because if this client used regular TCP, it would obtain only half of the shared bottleneck. This happens because each subflow increases its congestion window independently of the other subflows. This is called an uncoupled congestion control.

Specific Multipath TCP congestion control schemes have been designed to solve this problem. Four of them are included in the Linux kernel implementation: the Linked Increase Algorithm (LIA) [76], the Opportunistic Linked Increase Algorithm (OLIA) [40], the Balanced Linked Increase Algorithm (BALIA) [58] and the wVegas delay based congestion control algorithm [11]. LIA is the default congestion control scheme, but users can opt for other congestion

control schemes. They measure congestion on each subflow and try to move traffic away from those with the highest congestion. To do so, they modify the congestion window during the congestion avoidance phase. The congestion information of all the subflows belonging to a connection is taken into account to control the increase rate of a subflow's congestion window during the congestion avoidance phase. They are so coupling the increase phase to the subflow's congestion control. This kind of congestion control schemes preserves fairness with regular TCP across the shared bottleneck.

Chapter 3

Measurement infrastructure

Before going in depth in our measurements and the results obtained, it is useful to have first a look at our available resources to perform them. This chapter is divided in five parts. The first one provides an high level overview of our measurement testbed. The second one is dedicated to the different available hardware used. The third one presents the Multipath TCP kernel that was installed on the Android devices. The fourth one gives additional details about our SOCKS proxy which is the core of our measurement testbed. The last one describes some tools that we specifically developed for custom measurements.

3.1 High level view

An overview of our measurement testbed with smartphones is presented here with the help of the figure 3.1 on the following page. In order to make smartphones Multipath TCP capable, we bring some modifications into the Android kernel. Those are explained more in details in the following sections. Smartphones have two different network interfaces: the cellular one and the WiFi one. The cellular network is provided by Proximus and supports 4G. Except in chapter 5 (measurement at medium scale), we have the control of the WiFi router. More details about them are given in section 3.2.3 on page 13.

Since only very few real servers are Multipath TCP capable, we set up a SOCKS proxy between smartphones and remote servers. This allows smartphones to transfer data using Multipath TCP to the proxy, which can then send the same data using TCP to the real servers. This proxy runs on a server which is described subsequently in details. Although the connection is labelled as TCP on figure 3.1 on the following page, it can be a Multipath TCP one if

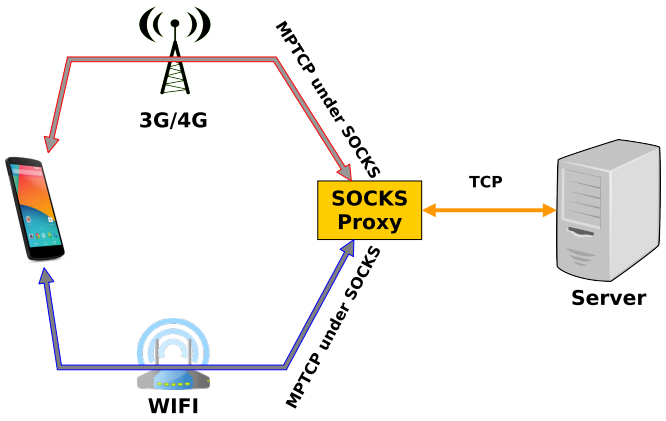


Figure 3.1: Our measurement testbed

the remote server is able to use it. Each Multipath TCP connection between smartphones and proxy is associated to one (possibly Multipath) TCP connection between proxy and real servers. The SOCKS proxy is explained in section 3.4 on page 15.

To analyse the network traffic, we capture network traces on the proxy thanks to `tcpdump` described in appendix A.3 on page III. For the automated measurements shown in chapter 4, we do the same on the smartphone. We then use analysis scripts (explained in section 3.5.1 on page 18) which take advantage of two main tools, `tcptrace` [50] and `mptcptrace` [32], to extract information from those traces. More details about these tools are given in appendixes A.1 on page I and A.2 on page II.

3.2 Infrastructure

To complete successfully our measurements, we had to set up various kind of hardwares.

3.2.1 Smartphones

Our smartphones were legacy Nexus 5 co-developed by LG Electronics and Google. These were the best ones on the market at the beginning of this study and certainly the most popular Nexus product[1]. The CPU is a Qualcomm Snapdragon 800 (2.26 GHz quad-core), 2 GB of RAM are available and it supports 2G/3G/4G LTE networks[2].

During all our tests, these smartphones were running Android KitKat 4.4 as operating system. We tried to not modify so much the smartphones. The main changes were made on the

[1] See The Verge's article: http://is.gd/GoFzN2
[2] See https://en.wikipedia.org/wiki/Nexus_5

kernel to support Multipath TCP as described in section 3.3. Two Android applications were required for our measurements to support Multipath TCP feature on all TCP connections on both cellular and WiFi interfaces for any servers: ShadowSocks and MultipathControl as respectively detailed in sections 3.4 on page 15 and 3.5.2 on page 22.

We used three Nexus 5 for the first part of our measurements described in chapter 4 on page 25: two for the tests and development and one dedicated to record the tests. For the second part detailed in chapter 5 on page 43, most smartphones have been set up to support Multipath TCP and given to people from INGI staff to generate real traffic. For the third and last part mentioned in chapter 6 on page 61, one Nexus 5 with Multipath TCP support and one without it have been used.

3.2.2 Server

During the entire period covered by the measurements, we had access to a dedicated server at OVH located in Roubaix (France). The selected product was a Kimsufi KS-2 with an Intel® Atom™ CPU N2800 running at maximum 1.86 GHz on two multithreaded cores (4 threads). It has 4 GB of RAM and has access to a 100 Mbits network with one IPv4 and one IPv6 dedicated addresses. The mean round-trip-time obtained with the `ping` tool from the UCL's wired network was about 7 ms. On this server, we ran different services isolated in different containers as explained in the appendix A.5 on page IV. The GNU/Linux operating system was a Debian Stable (Wheezy 7) with the latest stable version of Multipath TCP (0.89.5). The kernel version was `3.14.33-mptcp`.

3.2.3 Routers

Because it can be interesting to launch our automated tests when controlling the network by adding delay, losses and shaping and being sure that no other people were using our WiFi access point, we set up a WiFi router between the university wired network and our smartphone dedicated to the recorded tests. This router, a TPLink WDR4900, has latest version of OpenWRT as operating system and WiFi was only used on 5 GHz frequency, since no other access point was using this frequency range in the neighborhood.

When studying in details the traffic when switching from wireless networks to cellular ones, we decided to set up several routers in the ICTeam buildings. We used routers running OpenWRT with WiFi 2.4 GHz and connected to the university wired network.

3.3 Android's Multipath TCP kernel

A Multipath TCP kernel for Android devices was of course needed for our measurements. It was not so easy to have this kernel for two main reasons:

- Android kernel is still based on the 3.4 version of the Linux kernel, but patches for enabling support of the latest stable version (0.89) of Multipath TCP are based on Linux Kernel Longterm Support release v3.14. Many changes have been applied in the TCP/IP stack between the two versions.
- Google kernel developers have also integrated many specific changes in this Android kernel and many modifications in the TCP/IP stack.

A previous port for the Nexus 5 mainly made by Gregory Detal was publicly available on Github[3]. Unfortunately, it was based on Multipath TCP 0.86 version. Many improvements were made since this old version. To get valuable results, a new port was needed. This complex job has been realised by Gregory Detal and Sebastien Barré last summer.

When experimenting this kernel at the beginning, we found that switching off a network interface on the smartphone were causing a reboot. Our debugging analyses allowed us to find the reason: it was due to the use of the `tcp_nuke_addr()` function. This function has been added by the Android kernel developers in order to destroy all TCP sockets on the given local address. This cleaning was automatically made when an interface was disconnected. It seems that this function has been added to save memory which is understandable on this kind of devices. But in our case, with Multipath TCP, this behaviour is not acceptable as we cannot kill all sockets linked to an interface. Indeed, on the Multipath TCP Linux implementation [52], a socket in the TCP/IP stack can be a regular TCP socket, a MPTCP sub-flow socket or a MPTCP meta socket. This function should be adapted to only destroy regular TCP sockets. Even if we lose both WiFi and cellular connections during a few milliseconds, it is still possible to create new subflows after having new IPs addresses in order to keep the connection alive. As the memory optimisation choice performed by the Android developers is less relevant with the Nexus 5 (thanks to its 2 GB of RAM), we decided to simply bypass the use of `tcp_nuke_addr()` function.

Since last summer, a few micro versions of Multipath TCP have been released. We backported all patches from version 0.89.0 to 0.89.5 except the modifications due to other changes in upstream TCP stack of recent Linux versions. We also backported one patch from Christian Pinedo from the 0.90 branch in order to improve active and backup subflow selection in the scheduler[4]. In total, we backported 24 patches (24 files changed, 434 lines insertions, 196 deletions).

We used Gregory's instructions[5] to compile our Android kernel. We kept all default options dedicated to Nexus 5 devices (aka `hammerhead`). The only exception was to enable the Multipath TCP support with all TCP congestion control algorithms and the Multipath TCP's path managers and schedulers available. Note that by default with the latest version of our kernel, we used LIA as TCP congestion control algorithm, `full-mesh` for the path manager and `default` (prefer path with lowest round-trip-time) as scheduler.

[3] See https://github.com/gdetal/mptcp_nexus5
[4] See https://github.com/multipath-tcp/mptcp/commit/f9ca33d
[5] See https://github.com/gdetal/mptcp_nexus5/wiki

3.4 SOCKS proxy-like with ShadowSocks

According to the Multipath TCP's website[52], it seems there are only a few dozens of Multipath TCP enabled public servers on Internet. Multipath TCP requires that both client and server support it. For our tests it was not possible to convince maintainers of all visited servers to support Multipath TCP.

To overcome this issue, we configured all smartphones to use a Multipath TCP capable proxy server for all its TCP connections as represented on the figure 3.1 on page 12. With this configuration, each TCP connection initiated by the smartphone is thus redirected to and terminated at the proxy server. This proxy server then establishes a regular TCP connection to the server. Thanks to this setup, the smartphone can use Multipath TCP over the cellular and WiFi interfaces while interacting with legacy servers. Our setup is similar to the one proposed in [62, 19] except that we use a SOCKS-like proxy called ShadowSocks: a light SOCKS-like proxy with several implementations for both clients and server sides. Some tweaks on the host machine have been applied as recommended by ShadowSocks developers[6]. Note that our proxy only supports IPv4; more details about the IPv6 issues are given in appendix C on page XI.

3.4.1 How SOCKS works

According to the SOCKSv5 specifications [44], the client has first to negotiate the authentication method with the server. Then the client sends a request before exchanging data: `CONNECT`, `BIND` or `UDP`. In our case, we will only focus on the `CONNECT` command because we will only analyse TCP connections started by the client.

```
+----+-----+-------+------+----------+----------+
|VER | CMD |  RSV  | ATYP | DST.ADDR | DST.PORT |
+----+-----+-------+------+----------+----------+
| 1  |  1  | X'00' |  1   | Variable |    2     |
+----+-----+-------+------+----------+----------+
```

Figure 3.2: Format of a SOCKS request as specified in the RFC1928 [44].

The format of a SOCKS request is shown on figure 3.2. This first data of a connection should contain: one octet for the version of SOCKS used (e.g. `5`), one octet for the command (e.g. `5` for `CONNECT`), one octet of reserved field, one octet to specify the type address (IPv4, IPv6), some bytes indicating the destination address (4 octets for IPv4, 16 for IPv6) and two octets for the destination port.

The client has first to initiate a new connection with this packet before being able to send and receive data. An application which would communicate with a SOCKS proxy has to be adapted to respect these specifications. As all the needed test applications were not

[6] See http://shadowsocks.org/en/config/advanced.html

supporting a SOCKS proxy, we had to use ShadowSocks client on Android to redirect all TCP connections to our ShadowSocks proxy.

3.4.2 Characteristics of ShadowSocks's traffic

In order to open a new connection with a remote server using ShadowSocks, the smartphone first opens a new Multipath TCP connection to our proxy with the classical exchange of `SYN`, `SYN/ACK` and `ACK`. Then, the ShadowSocks client immediately sends to the proxy the `CONNECT` SOCKS request containing information needed by the proxy to open a new connection from itself to the remote server. Note that this last connection is either a Multipath TCP one if the remote server is Multipath TCP capable, or a TCP one.

It's interesting to also note that both ShadowSocks client and server do not seem to respect the SOCKS specifications [44] as shown on figure 3.2 on the preceding page. Indeed, the `CONNECT` requests only contain the command, the destination IPv4 address — IPv6 is not supported, more details in appendix C on page XI — and the destination port for a total of seven bytes instead of ten. Encryption is supported between ShadowSocks clients and servers for all exchanged data, even the first one. We choose the simplest encryption scheme to reduce overhead: a simple translation table is used which means that for each byte, a corresponding one is sent. It also means that the number of bytes sent and received after these seven bytes of `CONNECT` request is equivalent to the number of bytes sent and received by the application. Moreover, unlike a SOCKS proxy through a SSH tunnel where only one TCP connection is used between the client and the server, a new connection will be created between the smartphone and the proxy for each TCP connection initiated by an application on the smartphone.

3.4.3 How all traffic is redirected to the proxy

Let's take a simple example of visiting `example.org` website with a basic web browser on the smartphone as shown in the figure 3.3 on the facing page. There are two mains requests initiated by the application in this process: a DNS request is needed to get the corresponding IP (in red on the left of this figure), then a HTTP request is sent to retrieve the HTML content (in blue on the right).

The DNS request

(a) First the application queries system's DNS client for the IP address linked to the `example.org` domain.

(b) This client sends a UDP DNS request to a DNS server. By default on Android it is one of the Google's public DNS servers (`8.8.4.4`). Thanks to some specific forwarding rules,

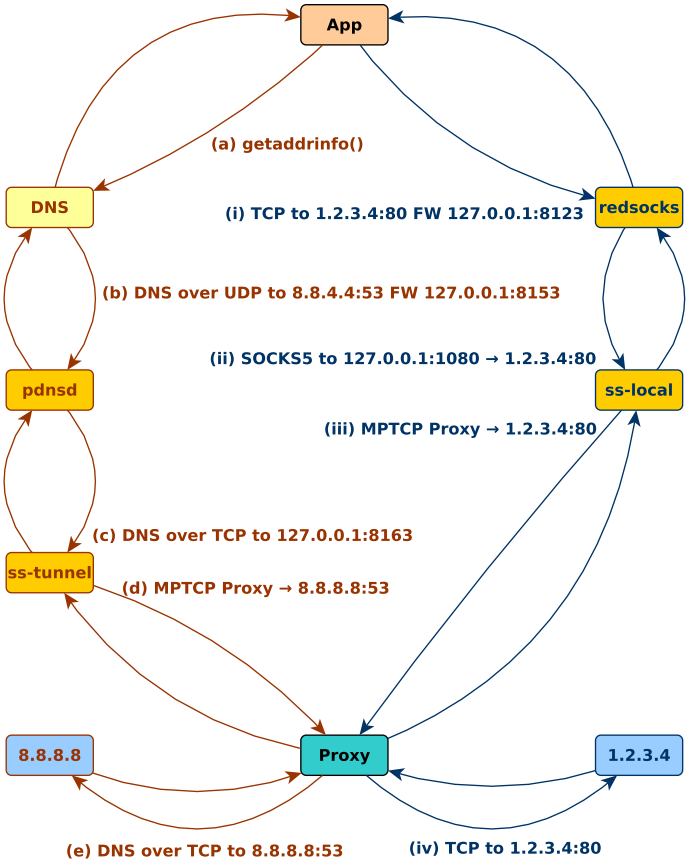


Figure 3.3: How all traffic is redirected to the proxy

all DNS requests are transparently redirected to a local DNS proxy called `pdnsd` [65] and controlled by ShadowSocks.

(c) `pdnsd` is configured to send all DNS requests via TCP. If a DNS record is not already cached, then a DNS request over TCP is sent to ShadowSocks's `ss-tunnel` daemon.

(d) This daemon initiates a new TCP connection to our ShadowSocks proxy at OVH in order to send this DNS request to another Google's public DNS server (`8.8.8.8`) specified by the ShadowSocks client.
Note that with the 2.5.8 version of ShadowSocks (used only for the automated tests in chapter 4), this daemon does not send to the proxy a first packet of seven data bytes with the already seen `CONNECT` command and an IP address. A special one with this time eleven data bytes and a hostname is sent: one for the command (4 and not 1), one for the length of the hostname (7), some bytes for the hostname (`"8.8.8.8"` which contains seven characters) and two for the port number (53). Since the IP remains constant, the hostname encoded in seven characters is not needed. Instead of sending

the seven bytes for the hostname, ShadowSocks can directly use the IPv4 address encoded in four bytes. Newer versions of this ShadowSocks client have reduced this overhead by sending the first packet of seven data bytes with the `CONNECT` command and the IPv4 address.

(e) Finally the proxy launches this DNS request over TCP and replies back the DNS answer to the client.

The HTTP request

(i) The application initiates a TCP connection to a specific IP and port (e.g. `1.2.3.4` on port 80). Again, thanks to the forwarding's rules, this request is forwarded to a `redsocks` daemon.

(ii) `redsocks` [25] is a transparent SOCKS redirector. It uses the destination TCP port and IP address of the original request in unauthenticated SOCKSv5 requests with ShadowSocks's `ss-local` daemon as described in section 3.4.1 on page 15.

(iii) `ss-local` is now able to start a new TCP connection with the proxy and send the right IP address and TCP port in the seven data bytes mentioned in the previous section 3.4.2 on page 16.

(iv) The proxy finally initiates a TCP connection to the given server (e.g. `1.2.3.4` on port 80) in order to let the client's application exchange data with it via this proxy.

3.5 Developed tools

Since some of our requirements are very specific (and others new to our knowledge), we had to develop specific software to fulfil them. In this section, we focus on our most important contributions. Note that all our public repositories are available on this page.

3.5.1 Analysis scripts

In the next chapters, we present the results of our measurements. To obtain them, we had to analyse the network traces to extract relevant information. This requirement was addressed to our analysis scripts, publicly available at [16].

Figure 3.4 on the next page is used to explain the architecture of our analysis scripts in the following paragraph. To start the analysis, we call the `analyze.py` script with the path to our Multipath TCP traces. These traces are first processed by `mptcptrace` [32] (step 1 in the figure). The outputs of this tool (CSVs, graphs) are processed by `mptcp.py` which collects all data we are interested in about Multipath TCP connections (2). Then, we give the same traces as input of `tcptrace` [50] (3). Files generated by this tool are then processed

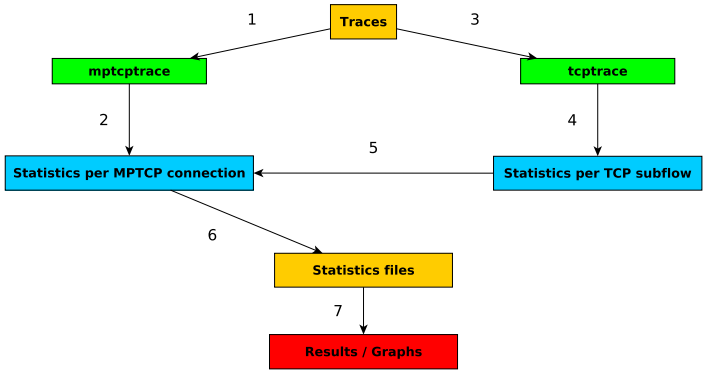


Figure 3.4: High level view of the analysis scripts for Multipath TCP traces

by `tcp.py` which allows us to collect more specific information about subflows at the TCP level (4). After that, still in `tcp.py`, we establish the match between TCP connections of `tcptrace` and Multipath TCP subflows of `mptcptrace` to merge statistics from both tools (5). All those data are then stored in statistics files (6) which allow us to produce quickly various graphs with the remaining analysis scripts (7). To illustrate our choice to store statistics in files, the analysis of all real smartphone traces studied in chapter 5 to generate statistics files lasts for nearly two entire days whereas the graphs generation from the statistic files takes less than 10 minutes. Note that our scripts are also able to process regular TCP traces; in that case, the analysis is restricted to the steps 3, 4, 6 and 7 of the figure 3.4.

Our analysis scripts contain more than 7500 lines of code in Python (without counting blank and comment lines) in 19 files[7].

Merging information

Since we want to have the maximum of information about connections, it is useful to merge statistics coming from `mptcptrace` with those coming from `tcptrace`. However, there is a mismatch between both tools. In particular, for Multipath TCP traces, a subflow seen by `mptcptrace` will be interpreted as a regular TCP connection by `tcptrace`.

To cope with that, we first analyse the traces with `mptcptrace` (in the file `mptcp.py`) and collect the start time and the duration of the Multipath TCP connection as well as the source IP address, the destination IP address, the source port and the destination port of the subflows. Then, we provide the same traces as inputs to `tcptrace` and identify the subflow the TCP connection belongs to thanks to the IP addresses, the ports and the start time. For short traces (as studied in chapter 4), relying only on the IP addresses and ports is sufficient. However, for very long traces (such as the ones analysed in chapter 5), it is not the case anymore since the same quadruplet can be reused after some time. To manage that, we

[7] This number and the following ones were computed by `cloc` command line tool.

also check if the starting time of the TCP connection (returned by tcptrace) fits with the Multipath TCP start time and duration. Note that it is feasible to have TCP connections detected by tcptrace which are not subflows of Multipath TCP connections. This is due to the fact that tcptrace also returns connections that are not complete, for instance the ones containing only a SYN followed by a RST. Additionally, it can be possible to intercept real TCP connections, i.e. connections that does not use Multipath TCP at all.

Structure of the statistics

To generate graphs, it is useful to have a look at the structure of the statistic objects. Those elements are either MPTCPConnection if the analysed trace is a Multipath TCP one (defined in mptcp.py) or TCPConnection if it is a regular TCP one (defined in tcp.py). Since the main protocol of interest is Multipath TCP, we will focus on the structure of MPTCPConnection but design choices remains the same for both objects. MPTCPConnection has three attributes: the ID of the connection, a dictionary of MPTCPSubFlow (or a single BasicFlow in the case of TCPConnection) and a dictionary called attr containing global information about the connection. To illustrate the content of the attr variable of MPTCPConnection, an example is shown in the listing 3.1.

Listing 3.1: Example content of the attr variable of MPTCPConnection.

```
{
  'destination2source':
  {
    'bursts': [(0, 14252, 0.160025, 1430498140.227634),
               (1, 54636, 0.182170, 1430498140.387996),
               (0, 1400, 0.0, 1430498140.604483),
               (1, 525064, 28.565089, 1430498140.604528)],
    'bytes': {'cellular': 537664, 'wifi': 14252},
    'bytes_mptcptrace': 525317,
    (...),
    'rtt_median': 48.602999999999994,
    'rtt_min': 2.0030000000000001,
    'rtt_samples': 436,
    'rtt_stdev': 60.449838352596807,
    'throughput_mptcptrace': 16234.924999999999
  },
  'duration': 29.118333,
  'source2destination': {(...)},
  'start_time': 1430498140.051284
}
```

Typically, it contains information computed by mptcptrace, except the bytes attribute in destination2source and source2destination, which comes from data computed by tcptrace. Most of them are direction related, except the start time and the duration of the Multipath TCP connection. destination2source refers to the exchange of data from the server to the client, where the client is the host that initiates the connection. Among others, we collect the number of bytes transferred at Multipath TCP level, various values for the round-trip-time at the Multipath TCP level and, if the connection is data heavy enough, the mean of the throughput at Multipath TCP level computed over 250 packets. This choice

of 250 packets is motivated to approximate as well as possible the actual throughput with application shaping (the instantaneous throughput is too variable and the throughput computed on the whole connection can be biased by application shaping). The `bursts` attribute can be used to compute the number of subflow switches per second.

Listing 3.2: Example content of the `attr` variable of `MPTCPSubFlow`.

```
{
  'daddr': '172.17.0.110',
  'destination2source':
  {
    'bytes': 14252,
    'bytes_data': 15652,
    'bytes_frames_retrans': 1486,
    'bytes_retrans': 1400,
    'frames_retrans': 1,
    'missed_data': 0,
    'packets': 15,
    'packets_outoforder': 0,
    'packets_retrans': 1,
    'reinjected_orig': {(755491846, 755490446): 2, (...)},
    'reinjected_orig_bytes': 25200,
    'reinjected_orig_packets': 18,
    'reinjected_orig_timestamp': [1430498140.604528, 1430498140.60454, (...)],
    (...),
    'rtt_99p': 81.309999999999988,
    'rtt_avg': 47.9,
    'rtt_from_3whs': 13.1,
    'rtt_max': 82.7,
    'rtt_median': 47.5,
    'rtt_min': 13.1,
    'rtt_samples': 2,
    'rtt_stdev': 0.0,
    'tcpcsm_retrans': [('1430498140.604483', 'RTO')],
    'timestamp_retrans': [1430498140.604483]
  },
  'dport': '8000',
  'duration': 27.966849088668823,
  'interface': 'wifi',
  'saddr': '130.104.236.212',
  'source2destination': {(...)},
  'sport': '57598',
  'start_time': 1430498140.038117,
  'type': 'IPv4',
  'wscaledst': '11',
  'wscalesrc': '6'
}
```

Some pieces of information are more specific to a subflow than to the entire connection. Those are then stored in the `attr` variable of the `MPTCPSubFlow`. An example of such information is given in listing 3.2. Except the start time and the duration, non direction related elements comes from `mptcptrace`. Such elements include the 4-tuple formed of IP addresses and ports, the version of IP used and the physical interface on which the flow was established. On the contrary, except reinjections information from `mptcptrace`, frames information from `tshark` and retransmissions seen by `tcpcsm` (described in appendix A.4 on page III), direction related elements are outputs from `tcptrace`. Among information computed at TCP level, there are

the number of bytes and packets (seen and retransmitted), the timestamp of retransmissions and various percentiles of the round-trip-time. Note that information related to the reinjection are not packets which are reinjections, but packets that were first send on this subflow and then reinjected on another one. In particular, we collect the timestamp of their reinjections, the number of bytes and packets those represent and the number of time a packet, identified by the range of sequence numbers it covers, is reinjected.

Using those statistics, we can then generate graphs to agglomerate them. A large part of the available scripts are dedicated to this usage, like `summary.py`, `summary_imc.py` or `macro.py`. In addition to the previously described scripts, they mainly rely on two Python libraries called `numpy` [21] for scientific computing and `matplotlib` [37] to generate graphs.

Optional processing

Additional features can be activated through options given to `analyze.py`. Among others, there is the identification of the interface used by looking at a MongoDB database (more details in section 3.5.2), rewriting the PCAP file to overcome some issues related to the loopback interface as explained in section 4.4 on page 30 or qualifying the type of retransmission. Some of these options require additional tools, such as `tshark` [75] to filter PCAP files, `tcpreplay`'s `mergecap` tool [41] to merge several PCAP files in one or `tcpcsm` [2] to give more information about congestion events on a connection.

Other scripts specific to the chapter 6 related to Streaming applications have been created. These scripts (200 lines in Python, 141 in Bash, without counting blank and comment lines) don't analyse PCAP files and are available in another public repository [7]. For instance, the results show a waveform of the sound produced by a streaming application and a map of the journey with indications about network changes. Those are also generated thanks to `numpy` [21] and `matplotlib` [37] in addition with `geotiler` [71] to create maps using tiles from a map provider like OpenStreetMap and `waveform` [38] to generate visuals for audio files.

3.5.2 An Android application to control Multipath TCP

Simply installing a new Multipath TCP-ready kernel is not enough if you want to use multiple network interfaces at the same time. Indeed, you need to configure routing tables as described on the Multipath TCP's website[8].

Automatically set routing tables

Four different approaches have been compared: *(i)* modifying Android's frameworks base code, *(ii)* modifying this same code with the help of the Xposed framework, *(iii)* deploying

[8] See http://multipath-tcp.org/pmwiki.php/Users/ConfigureRouting

with the kernel image a new low level service that reacts to network changes and *(iv)* using an Android application which is notified by the Android's framework of network changes.

(i) Directly adapting Android source code should give better results as we will be able to control the operating system's behaviours. On the other hand this approach requires a good study of the Android's frameworks base code and the need to rebuild Android's images and reinstall them on all smartphones. Furthermore, this change will be specific to one version of the AOSP Android's framework.

(ii) Xposed[9] is defined as *a framework for modules that can change the behavior of the system and apps without touching any APK*. In theory, we could intercept call of methods and modify their behaviour. Unfortunately, we didn't have time to study in details the Android's frameworks base code and this Xposed framework.

(iii) Deploying a new low level service should give us sufficient control by being notified before other by Linux's `netlink` when a route or an IP has been added or deleted. This service has to be shipped with the kernel image. It means it has to be adapted to work with our kernel and to allow some controls by the user, etc.

(iv) Finally we chose the last approach: a dedicated Android application made by Gregory Detal and called MultipathControl was available[10]. This application was already working by modifying routing tables when IP addresses change and by sending requests to the Android Framework in order to keep the cellular interface enabled. We used it for the first part of our work without any problem but we had to adapt this application as this one was not ready to be used by real users.

Improvements made on this application

Some features were required before giving this application to real users. To do so an Android service has been added to keep this application on even if it is running in background. This service is also automatically launched at startup. IPv6 support on Android has also been disabled with this application as explained in appendix C on page XI. Thanks to these modifications and the ShadowSocks proxy, users are able to use Multipath TCP via both WiFi and cellular interfaces without doing anything else.

After having refactored a bit the code and fixed small bugs, a few options have also been added such as changing the default route via the cellular interface, setting subflow using cellular interface as a backup subflow and changing the default TCP congestion control algorithm. But the most important change in terms of code was about various data collection. Indeed collecting information on network changes can help us to analyse in more details how the devices reacted on handovers[11] such as what were the conditions during those actions, which

[9] See http://xposed.info

[10] See http://multipath-tcp.org/pmwiki.php/Users/Android

[11] *Handover* term is used here to specify the action when transparently switching from one to another network.

IPs were used at that time, etc. To collect all these data, we reused a part of the Carmen Alvarez's Network-Monitor project [4] released under the 2.0 Apache License. Each hour, collected data were sent in `JSON` format to a custom HTTP/REST server [17] described in appendix B.3 on page VIII.

Because this application collects many private data[12] and has root privileges, we wanted to release the code under an Open Source licence. With Gregory Detal's agreement, this application has been released under a GNU/GPLv3 license and is freely available on Github [9]. This MultipathControl application is composed of 2182 lines of code (without counting comment lines). HIPRIKeeper application has also been created just to let the cellular interface activated on any smartphone. It is made of 379 lines of code (without counting comment lines).

Limitations of this application

By choosing this solution with an Android application, we were aware that it was not the optimal solution despite of its advantages. The main drawback is that just after being attached to a new WiFi access point, the Android framework believes that the kernel has stopped all connections due to Android kernel's `tcp_nuke_addr()` function previously described in section 3.3 on page 13. As a result, this Android framework starts disabling the cellular interface. It had to be done because by default Android has been configured to use only one interface, preferring its WiFi one to the cellular one. This is justified as two interfaces cannot be used at the same time for the same connection with regular TCP. Then it's obvious that only the WiFi interface is kept if attached to a WiFi access point mainly because it's free of cost (most of the time) and there is no need to keep the cellular radio on if this interface is not used.

This behaviour implies that TCP connections could be stopped if the MultipathControl application is not fast enough to stop the Android framework from disabling the cellular interface. But even if this interface is disabled for a few milliseconds, it is absolutely not a problem for Multipath TCP. Indeed, by closing this TCP connection, only one Multipath TCP subflow will be stopped and not the entire connection. This is why a new Multipath TCP subflow will be created just after having re-enabled the interface to keep the connection alive. In other words we could have better results on handovers with a better integration of Multipath TCP in Android framework but our objective was to obtain promising results related to handovers. Then, a future work would be to implement this integration.

[12] Unfortunately as most popular applications do.

Chapter 4

Automated measurements

Before going in depth with experiments, a first step would be to see the behaviour of Multipath TCP in a *controlled environment*. By this, we mean that we would make experiments relative to a reference case, and then voluntarily apply changes to see their impacts. Those reference cases are implemented as scenarios with different applications, covering various network activities. With our automated test framework, we can then control the protocol used by the smartphone (either TCP or Multipath TCP), interfaces used and performances of networks (in particular the WiFi access point).

Researchers have analysed various aspects of Multipath TCP on smartphones. While we used real smartphone applications, the earlier studies mainly relied on bulk transfers that do not represent real smartphone usage. Arzani et al. model the performance of Multipath TCP in [5] and show some capture effects for the initial subflow. However, the capture effects that they observed is different from the impact of the default route that we measured. Chen et al. analyse in [12] the performance of Multipath TCP in WiFi/cellular networks by using bulk transfer applications running on laptops. Deng et al. compare in [18] the performance of single-path TCP over WiFi and LTE networks with the performance of Multipath TCP on multi-homed devices by using active measurements and replaying HTTP traffic observed on mobile applications. Their measurements show that Multipath TCP provides benefits for long flows but not for short ones. For short flows, they show that selection of the interface for the initial subflow is important from a performance point of view. Ferlin et al propose in [27] a probing technique low performing paths and evaluate it in wireless networks.

Some previous works have also been done related to the use of proxies with smartphones. Tachibana et al. [69] have developed a Multipath transfer solution based on SCTP with a proxy

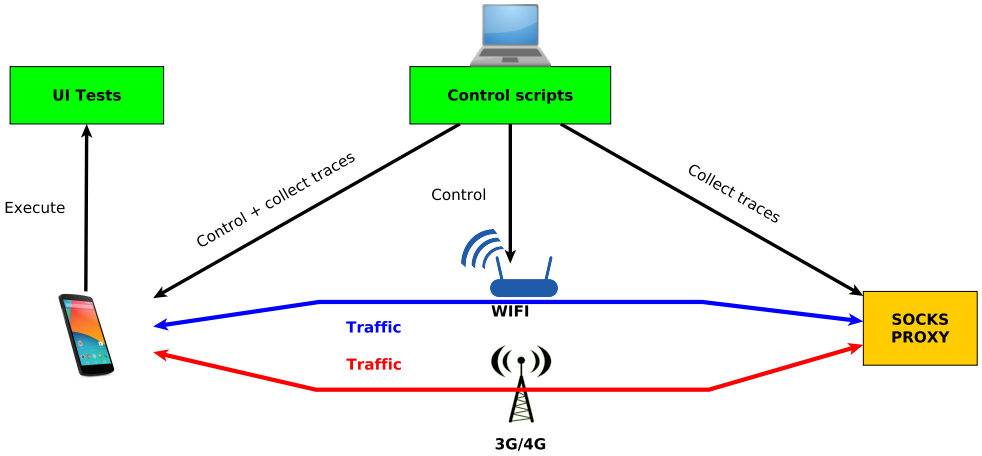


Figure 4.1: High-level view of the automated test framework

between the smartphones and the servers. Choffness et al. [13] analyse proxy behaviours and how transparent web proxies interact with HTTP traffic in four major US cell carriers.

In the remaining of this chapter, we first present the methodology followed to perform our measurements. Then, we describe the scenarios and the applications studied. After discussing some issues with analysing the loopback interface and briefly giving some details on the version of Multipath TCP on the smartphone, we show the performances of regular TCP with smartphones. Afterwards, we measure the impact of using Multipath TCP on studied applications. Finally, we conclude with the key lessons of those results.

4.1 Automated test framework

In this chapter, we describe results obtained by collecting a large amount of data in the most reproducible way. To obtain them, we developed an automated test framework. An overview of its operations is shown on figure 4.1. On a computer, we run scripts that monitor our system. In particular, those control the smartphone to launch UI tests under various conditions such as Multipath TCP mode, interfaces used by the smartphone, WiFi shaping and much more. Once executed on the smartphone, our UI tests generate network traffic between the smartphone and the proxy. This traffic is then captured both on the smartphone and the server to allow further analyses. Overall, our automated framework contains 1602 lines of source code in Python, 1933 in Java and 585 in Bash (without counting comment lines). This is publicly available at [8]. In the remaining of this section, we detail the two main part of the framework: the UI tests and the control scripts.

4.1.1 Android's UI tests

The first one contains UI tests that mimic user interactions with the smartphone applications to produce a given high-level scenario. Those scenarios can either be representative use cases of the studied applications or be used as tools to configure the smartphone. In both cases, those are implemented in Java using the MonkeyRunner Android UI testing tool [31]. Those tests are designed to face unusual situations, such as the failure of the device, a loss of connectivity or unusual behaviour of the application (application wanting the user to rate it, etc.). In order to have coherent results along days mainly due to changes in the applications, we stopped updating them on November 15th, 2014. The corresponding versions of studied applications are detailed in [8]. Note that before launching any test, caches of all applications are emptied. This prevents optimisation of applications based on previous runs (such as storing files that need to be reloaded each time) and makes the tests more reproducible.

4.1.2 Controlling execution of UI tests

The second one contains Python and Bash scripts that allow a computer to control the smartphone. It was designed to be reusable for other projects with smartphones but also to be as modular as possible, i.e. parameters can be defined in order to easily change the behaviour of the tests. Moreover, the framework is able to cope with multiple unexpected events, such as UI test failures or situations caused by the unreliability of the smartphone.

All settings are defined in `lt_settings.py` file. It is possible to override some of them by passing in argument another Python script file which contains all variables which will replace default ones. For instance we can easily change settings related to Multipath TCP, e.g. path manager, scheduler, backup mode, TCP Congestion Control algorithm ; to the router, e.g. delay, losses, shaping ; to the smartphone, e.g. enable or disable interfaces, change default route, change proxy method, etc.; to the environment, etc. A script to launch a Python console is also available to get quickly data or to do some tests.

These scripts are also responsible for collecting data and starting network traffic captures on both the smartphone and the proxy. Indeed, we used `tcpdump` [70] on both smartphone and proxy to have a full view of what is going on. Like this, all sent packets are seen, and we can observe, for instance, TCP retransmissions due to lost packets. On the smartphone, two files are generated. The first one contains all traffic going on the loopback interface (`lo`) and has its name suffixed by `_lo`. This is especially useful to easily see which servers the application contacts and what is happening on it. It was also used for debugging purpose, as described in section 4.4. The second one contains all traffic seen by the actual interfaces, i.e. the WiFi one (`wlan0`) and the cellular one (`rmnet0`). Its name is either suffixed by `_any` (if both interfaces are up) or by the name of the interface (if only one is up). This allows to see the actual traffic exchanged between the smartphone and the server. On the server, only one file is generated and captures traffic going on its actual interface. Note that although statistic files return various values, an additional care must be taken when the trace was taken in the

receiver direction of the data stream. Indeed, round-trip-times do not reflect the reality and information about retransmissions are biased (losses are not seen, etc.).

4.2 Methodology

To perform our measurements, we mainly rely on our automated test framework as described in section 4.1 on page 26. With it, this would be sufficient to perform regular TCP measurements, since applications work under this protocol. However, since we are interested in results with Multipath TCP, additional setup is required. Indeed, although our smartphone is Multipath TCP capable, this is not the case for remote servers that will be contacted by the studied applications. To cope with that, we installed a SOCKS proxy, and configured the smartphone to enforce all connections to go through the proxy which is Multipath TCP capable as described on section 3.4 on page 15. For each Multipath TCP connection between the smartphone and the proxy corresponds a TCP connection between the proxy and the remote server. Like this, the smartphone can use Multipath TCP without any change on the server side. The choice of SOCKS is motivated by its slight overhead, though it is not zero.

As shown on figure 3.1 on page 12, the smartphone has connectivity with two different interfaces. The WiFi network is provided by a router with a `802.11n` interface on the 5 GHz frequency band with a bit rate of 65 to 72 Mbits. The router was connected with a 100 Mbits link to the university network. In order to avoid possible interferences, we ensured that no other WiFi network was emitting in the same range of frequency in the building. The cellular network is a commercial one and is provided by Proximus, the biggest network communication company in Belgium. The smartphone can either be connected using the 3G or the 4G. The test scenarios were run in a random order on each day to prevent time correlation between the results and the time tests were launched. Note that all the tests were performed during the night to prevent interferences caused by other people on networks.

To obtain our results, we rely on our analysis scripts described in section 3.5.1 on page 18.

4.3 Application studied

This section describes the scenarios used to represent a relevant usage of smartphones. The studied applications cover various usage, from cloud storage (Dropbox, Google Drive) to web browsing (Firefox), passing through music (Spotify) and video streaming (Dailymotion, Youtube) and social networking (Facebook, Messenger). Even if scenarios are based on applications, the focus is not on applications themselves. The selection was first based on the kind of traffic applications generate, then on their popularity on the Google Play Store (in September 2014). Moreover, we also checked if our applications use TCP and not other protocols like UDP.

Our scenarios can be split into two categories: *upload intensive* scenarios (the smartphone mainly sends data to the server) and *download intensive* scenarios (the smartphone mainly receives data from the server). Note that we limit the duration of tests to 120 seconds, in order to have sufficient time to run tests during the night.

4.3.1 Upload intensive scenarios

We first consider two interactive applications : Facebook and Messenger. We expect these applications to send small volumes of data. With the Facebook application, our test first updates the news feed, it then writes a new status, takes and shares a new photo with a description and finally performs a new check out status. This scenario is repeated three times per test. Facebook only uses HTTPS during this test. With Messenger, it sends a text message, then puts a smiley and finally sends a new photo. This scenario is also repeated 3 times for each test and all data are sent over HTTPS.

Then we consider two cloud storage applications : Dropbox and Google Drive. With Dropbox, the test creates a fresh file containing 20 MB of purely random data and uploads it. It always uses random data to prevent Dropbox from compressing it [22]. If the upload was fast, it creates and sends a second one. All the data transfers are done above HTTPS. The same high-level scenario is used with Google Drive.

4.3.2 Download intensive scenarios

Our second class of applications contains the ones that mainly receive data. Our first test uses Firefox to browse the web always from a cleaned session. It simply retrieves the main page of the top 12 Alexa web sites. Firefox uses mainly ports 80 (HTTP) and 443 (HTTPS). The tested version was Firefox Beta 33.0.

Our second application is Spotify. This is a music delivery application. The test plays a new music (shuffle play feature) for 75 seconds. The music files are retrieved over HTTP but the application also uses HTTPS and TCP connections on port 4070 (used for Trivial IP Encryption).

Finally, we consider two popular video streaming applications: Dailymotion and Youtube. These applications are used to stream movies from cloud servers. They both measure the available bandwidth and adapt the requested video bit rate accordingly [64, 1]. With Dailymotion, the test plays three different videos in the same order and watches them for 25 seconds. Those videos are available in HD and we configured the application to fetch the best possible quality even when using cellular networks. Dailymotion uses HTTP and HTTPS. The same kind of test is performed with the Youtube application. In this case, only HTTPS is used and the video seems to be transferred using two TCP connections at the same time that evolve quite similarly.

4.4 About loopback interface

As explained in section 4.1 on page 26, our automated test framework is able to collect traces on loopback interface (`lo`) and actual ones (`rmnet0`, `wlan0`) on the smartphone. The main advantage of listening to the `lo` interface is that we can see the actual traffic sent by the applications to the servers, without any SOCKS wrapping. This means, amongst others, that we can know the IP address and the port of the remote servers. However, we face two main issues in analysing this interface.

First, we observe only unidirectional connections in the collected PCAP files, i.e. packets going either from the smartphone to the servers or from the servers to the smartphone. This is a consequence of the use of the RedSocks proxy (included in the ShadowSocks client). Indeed, all the upload traffic is directed to the local host on port 8123, whereas the download traffic comes from actual IP addresses and ports of remote servers. To cope with that, we leverage our analysis scripts by adding an option to rewrite the PCAP file with `tshark` [75] and `mergecap` [41], such as the connections become bidirectional. The merge can be done based on the ephemeral port number used by the application, since the number of the connections and the time of the capture (less than 5 minutes) is small enough to assume that the ephemeral port will not be used twice in the same PCAP file.

Second, we note that the IP address and the port indicated in the packets of this loopback interface and received by the smartphone from the servers do not reflect the actual path followed by the packet. Therefore, the loopback interface cannot be used to monitor the use of the different interfaces. Moreover, events such as reinjections and retransmissions are present on actual interfaces, but are hidden for the loopback one. This is why the remaining of our analyses is made on the `rmnet0` and `wlan0` interfaces. Note that it is still possible to collect the real IP addresses and ports of the remote server thanks to the SOCKS command sent by the smartphone to the proxy. This is explained in details in section 3.4.2 on page 16. However, those traces captured on the loopback interface were still useful to detect abnormal behaviours of the smartphone. Indeed, we observed strange results with some traces captured on `wlan0` and `rmnet0`, and we noticed that their corresponding `lo` traces were empty. Thanks to them, we were able to filter such traces.

4.5 Multipath TCP on smartphone

Several backports of the Multipath TCP kernel on Android smartphones have been released in the last years. However, these ports were often based on old versions of the Multipath TCP kernel. For this work, we rely on a backport of the latest version 0.89.5 of the Multipath TCP Linux kernel [52] on a Nexus 5 running Android 4.4.4 with some modifications, as explained in section 3.3 on page 13. The Multipath TCP kernel controls the establishment of subflows on the available interfaces thanks to a path manager. We use the Full Mesh path manager that creates a subflow over all network interfaces for each established TCP connection. The

Multipath TCP scheduler studied is the default one, i.e. it sends data on the available subflow having the lowest round-trip-time.

All the popular smartphone applications use TCP to interact with servers managed by the application developers. As of this writing, it is impossible to convince them to install Multipath TCP on their servers. To overcome this issue, we configured the smartphone to use a Multipath TCP capable SOCKS proxy server for all its connections, as explained in section 3.4 on page 15. Each connection initiated by the smartphone is thus redirected to, and terminated at, the proxy server. Thanks to this setup, the smartphone can use Multipath TCP over the cellular and WiFi interfaces while interacting with legacy servers.

This setup allows us to capture all the packets sent by both the smartphone and the SOCKS server. We captured more than 85,000 connections over about 1200 tests conducted in February and March 2015 carrying more than 15 GBytes of data.

4.6 Single-path measurements

Before analysing the behaviour of Multipath TCP, we first take a look at classical single-path measurements to see the traffic generated by the smartphone with a WiFi access point. We also characterise the performances of both WiFi and cellular interfaces.

4.6.1 Behaviour of TCP connections with WiFi

In order to analyse Multipath TCP, it is interesting to take first a look at the behaviour of the applications with classical TCP. Those are not homogeneous; they present various different characteristics in terms of number of connections, duration of the connections or exchanged bytes. Following results were obtained using only the WiFi interface (`wlan`) with TCP without shaping. A first overview of the TCP connections established by applications is shown on figure 4.2 on page 33. Each point on that figure corresponds to a TCP connection. The x-axis (in logarithmic scale) shows the duration in seconds of the connection and the y-axis indicates the total number of bytes exchanged (in both directions). Aggregated data per application is also shown on table 4.1 on the next page. Clearly, we see that Firefox is the application that uses the largest number of connections (those represent 63.9% of all connections). This is quite expected since our scenario contacts the top 12 Alexa web sites. Dropbox (31.75%), Youtube (29.7%), Drive (19.9%), Dailymotion (9.6%) and Spotify (4.96%) are the applications that exchange the largest number of bytes (in any direction). However, we observe that Facebook has some connections that last more than 100 seconds, but those are quite light in terms of bytes (no connection exceeds the megabyte).

Although we expect to encounter various network profiles with the different applications, the behaviour of Firefox is quite surprising. Indeed, at the bottom of the figure 4.2 on page 33, we observe a line composed of hundreds of Firefox's connections that last up to ten seconds but that only carry seven bytes of data. After investigation, this behaviour can be explained by

Application	# connect.	Bytes smart. to server	Bytes server to smart.	# tests
Dailymotion	76.0	22,202.4	12,246,855.1	5
Drive	17.0	21,601,411.4	3,821,191.2	5
Dropbox	21.5	40,519,111.25	58,240.5	4
Facebook	30.8	503,166.4	331,780.2	5
Firefox	440.2	162,958.0	39,895,76.6	5
Messenger	22.33	168,250.33	32,039.33	3
Spotify	42.4	44,665.2	6,288,466.8	5
Youtube	38.8	247,342.6	37,726,634.2	5

Table 4.1: Means of high-level characteristics for each studied application.

two factors. The first one is that Firefox preventively opens new TCP connections, but these are sometimes never used (i.e., the connection is open but no data bytes are exchanged). The second one is the SOCKS proxy, and is related to the seven bytes needed to allow the proxy to open the connection with the remote server, as explained in section 3.4 on page 15. Indeed, when Firefox tries to connect to a web site (let say `www.example.org`), the smartphone sends a DNS request over a TCP connection to the proxy. Upon reception of the answer, the ShadowSocks client opens a new connection to the proxy and then sends the `CONNECT` request (which is seven bytes long) needed to establish the connection between the proxy and the remote server.

Still on figure 4.2 on the facing page, there is a fog of Firefox's connections that last up to one second and exchange hundreds of bytes. Observing the timing of those connections, those correspond to DNS requests and responses on top of TCP: around 30-40 bytes from the smartphone and around 150 bytes to the smartphone.

Considering all applications together, connections can be categorised into three groups: *(i)* short connections (less than one second) carrying a relatively small amount of data, *(ii)* long connections carrying most of the data, and *(iii)* long connections carrying a small amount of data. Still taking the TCP connections with only `wlan` interface enabled without shaping, we observe that 74.05% of the connections last less than one second, but they only carry 1.14% of all data bytes. In the remaining long connections, 32.19% carry more than 10 KB of data and represents 98.58% of the overall data volume. The remaining 67.81% of the long connections exchange less than 10 KB of data, which represents only 0.28% of data exchanged. This tends to match many other measurement studies that have identified that most TCP connections are short and most of the traffic is carried by a small fraction of all TCP connections [30].

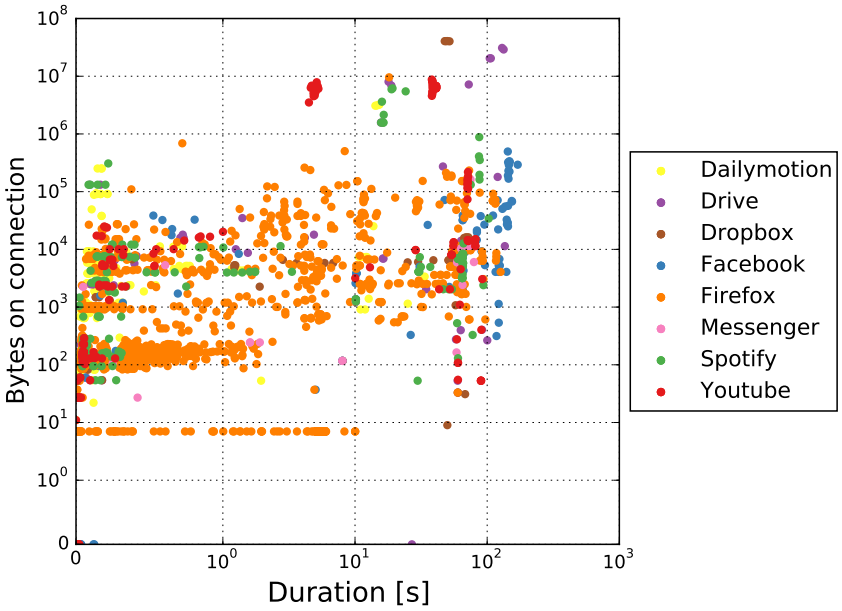


Figure 4.2: Smartphone applications present different network profiles.

4.6.2 Characterisation of interfaces

The round-trip-time (RTT) is one of the key factor that influences the performance of TCP connections. We used `tcptrace` [50] to compute the average round-trip-time for each of the captured TCP connections. Figure 4.3 on the next page provides the CDF of the round-trip-times measures among all the TCP connections used in the upstream (data sent by the smartphone) and downstream directions. The 4G network exhibits a round-trip-time in upstream with a median of 42.6 msec and a mean of 50 msec. In the downstream direction, the median round-trip-time increases up to 38.1 msec and a mean of 41 msec. On the WiFi network, around 60% of the connections have a round-trip-time shorter than 15 msec. Unsurprisingly, there is some bufferbloat[1] on the cellular network, mainly in the upstream direction, but the bufferbloat remains reasonable compared to other networks [28].

RTTs presented on figure 4.3 on the following page shows expected results, i.e. WiFi has the shortest RTT, then 4G and finally 3G. To confirm those results, we also generate the CDF of the RTT measures using Multipath TCP with only one interface up on figure 4.4 and we expect to obtain same results. Although both 3G and 4G keep the same RTT (which is expected), the results for the mean RTT of the WiFi interface are surprising. However, looking at the perceived RTT using both WiFi and cellular (3G or 4G) interfaces with Multipath TCP gives very similar results to the ones perceived with regular TCP with only one interface up (like figure 4.3 on the next page). The reason of such results for Multipath TCP with only `wlan` up is unclear for us and the academic staff.

[1] *Bufferbloat*: when too much packets are buffered on the network, it causes high latency, delay variation and then can reduce the overall network throughput.

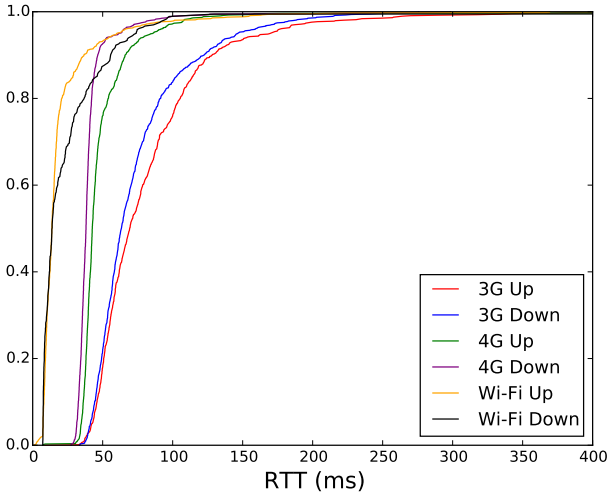


Figure 4.3: CDF of the mean Round-Trip-Time of TCP connections with one interface up.

Interfaces also present different bandwidths. During the night, we observe for 3G a upload bandwidth of around 2 Mbits and a download one of around 20 Mbits. Those number increases up to 15 Mbits in upload and around 60 to 80 Mbits in download for 4G. For WiFi, we obtain around 60 Mbits in both directions.

4.7 Multiple-paths measurements

The previous section has shown that our scenarios cover various utilisation of TCP. In this section, we now enable Multipath TCP (with both interfaces up) and rerun the tests to understand how our eight applications interact with Multipath TCP. Note that we didn't encounter any incompatibility with the applications and Multipath TCP, i.e. if remote servers were Multipath TCP capable, no changes would be needed on the applications.

Multipath TCP can be used in different modes [54] on smartphones. For our measurements, we focus on a configuration where Multipath TCP tries to pool the resources of the cellular and the WiFi interfaces simultaneously since the handover and backup performance have already been studied in [54].

4.7.1 Distribution of traffic on both interfaces

When a 4G and a WiFi interface are pooled together it is interesting to analyse which fraction of the traffic is sent over which interface. With the Multipath TCP implementation in the Linux kernel [52], this fraction depends on the interactions between the congestion control scheme, the packet scheduler, the underlying networks and the application.

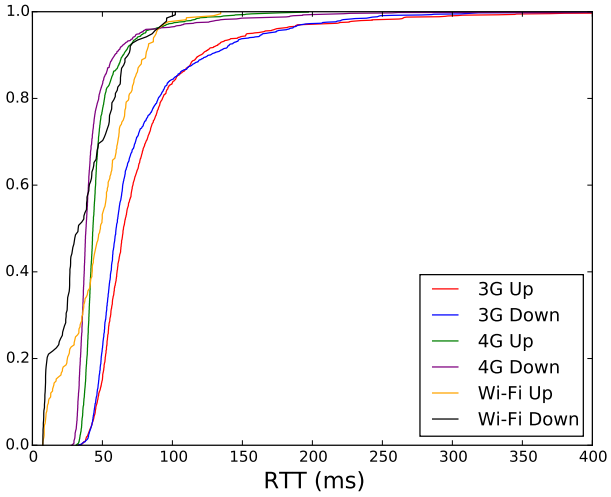


Figure 4.4: CDF of the mean Round-Trip-Time of Multipath TCP connections with one interface up.

On figure 4.5 on the following page, each point corresponds to one Multipath TCP connection, and the x axis indicates the number of bytes transferred by this connection from the smartphone to servers. We observe that 92.4% of all the connections only use the WiFi interface, but these connections only carry 1.1% of all data bytes.

There are several factors that explain why Multipath TCP does not use the cellular network for these connections. The first factor is the configured `default` route. When an application initiates a connection, Multipath TCP sends the `SYN` over the interface with the `default` route, in our case the WiFi interface. This is the standard configuration of Android smartphones that prefers the WiFi interface when it is active. If the Multipath TCP connection is short and only transfers a few KBytes or less, then most of the data fits inside the initial congestion window and can be sent over the WiFi interface while the second subflow is established over the cellular interface.

An example of application pushing data as fast as possible while opening a new subflow on the other interface is shown on figure 4.6 on page 37. The opening of a new subflow is a 4-way handshake. The smartphone first sends a `SYN` with the `MP_JOIN` option containing the identifier of the Multipath TCP connection we want to attach this subflow. A process similar to opening a regular TCP connection occurs, at the difference it needs a fourth packet (the last `ACK` between the proxy and the smartphone) and no data can be sent before reception of this last `ACK`. This is needed to authenticate the end-hosts, as explained in section 2.2.2 on page 6. If the application pushes data as fast as possible, it can send two congestion windows before the establishment of the other subflow (since it takes two RTTs on the other interface), which can be sufficient to transfer all data for small connections. Furthermore, the round-trip-time over the WiFi interface is shorter than over the cellular interface. This implies that, as long as the congestion window is open over the WiFi interface, Multipath TCP's RTT-based

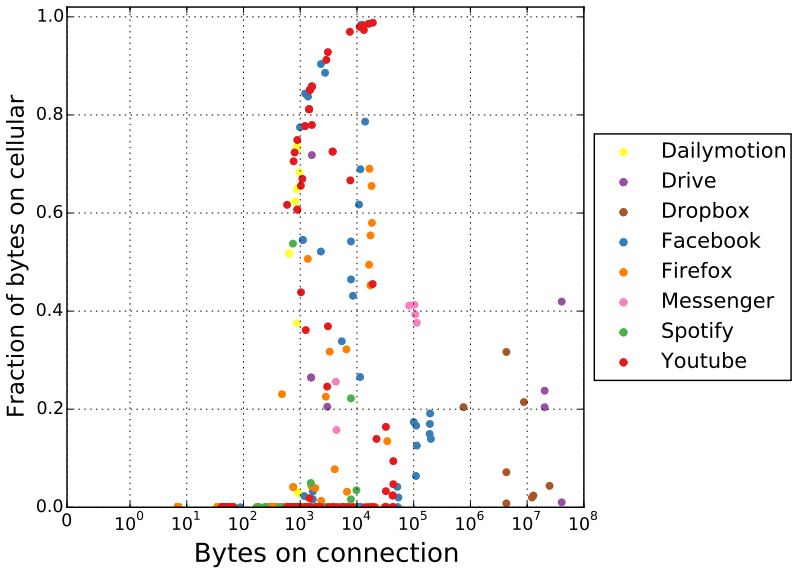


Figure 4.5: When the default route points to the WiFi interface, Multipath TCP mainly uses this interface for the short connections.

scheduler [55] prefers to send packets over the WiFi interface. Indeed, 94.88% of connections in that case have a better average RTT on WiFi than 4G (this percentage is 75.44% if the maximal RTT seen is considered).

Those factors explain why data on the short connections is exchanged only over the WiFi interface. We experimentally verified this by performing the same set of measurements with the default route pointed to the 4G interface. Figure 4.7 on page 38 shows that when the default route points to the cellular interface, most of short connections still use this one (as annotated as 1), but it concerns only 53.22% of all connections. It seems that even if cellular is the default interface, connections still mainly use WiFi, even for connections exchanging less than 1 KB. This occurs for connections that do not push data as fast as possible data. If the connection lasts more than two RTTs of the original flow, Multipath TCP has enough time to establish the second subflow. The packet scheduler will then select the flow with the lowest RTT — 82.74% of all connections have a WiFi flow with a lower maximal RTT than the cellular one.

This explains the bottom of figure 4.7 on page 38 (annotated as 2): a set of Firefox connections transfer fewer than 10,000 bytes always nearly exclusively on the WiFi interface. A closer look at the packet traces reveals that these connections are part of the connection pool managed by Firefox. This behaviour does not happen with other applications. When Firefox creates a connection in the pool, the initial handshake and the SOCKS command to our SOCKS server are sent. These packets are exchanged over the cellular interface and Firefox does not send immediately data over the established connection. This leaves enough time to

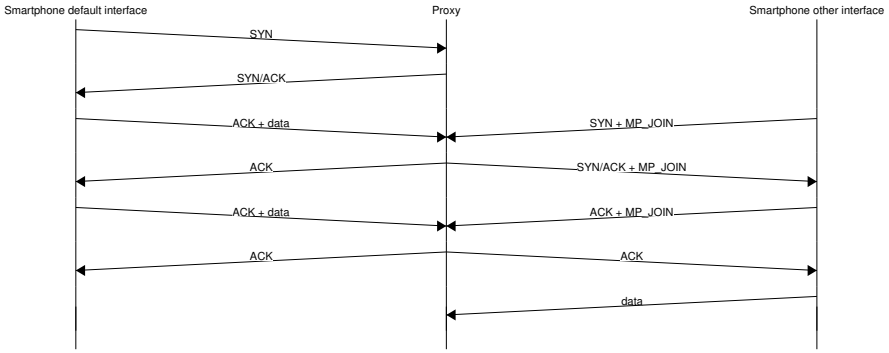


Figure 4.6: Time sequence diagram showing the smartphone opening a new subflow (assuming both interfaces have the same RTT).

Multipath TCP to create the subflow over the WiFi interface and to measure its round-trip-time. When Firefox starts to transmit data over such a connection, the RTT-based scheduler used by Multipath TCP prefers the WiFi subflow and no data (except the initial SOCKS command) is sent over the cellular subflow.

When the applications push more data over the Multipath TCP connection, the distribution of the traffic between the cellular and the WiFi interface also depends on the evolution of the congestion windows over both subflows. If the application pushes data at a low rate, then the packet scheduler will send it over the lowest-RTT interface (WiFi in our case). However, this can be fragile. If one packet is lost, then the congestion window is reduced and the next data might be sent over the other interface. If the application pushes data at a higher rate, then the congestion window over the lowest-RTT interface is not large enough and the packet scheduler will send data over the second subflow.

4.7.2 Multipath TCP's behaviour with shaped networks

An important benefit of the resource pooling capabilities of Multipath TCP is its ability to adapt to various networking conditions. When a smartphone moves, the performance of the WiFi and cellular interfaces often vary. Previous work with bulk transfer applications has shown that Multipath TCP can adapt to heterogeneous networks having different bandwidths and delays [56]. Our measurement framework also allows to explore the performance of smartphone applications under various network conditions.

As an illustration, we analyse the packet traces collected when the smartphone is uploading with Dropbox. The figure 4.8 on page 39 shows the means of the throughput T^* on connections uploading at least 1 MB. T^* is computed at each packet over the last 250 packets at Multipath TCP level. For instance, if a connection carries 500 packets, there will be 251 points for T^*. This is done to minimise the effect of the shaping made by the Dropbox application, which sends data by blocks of 4 MB with some idle time between them. We first consider a WiFi access point attached to a DSL router having 1 Mbps of upstream bandwidth

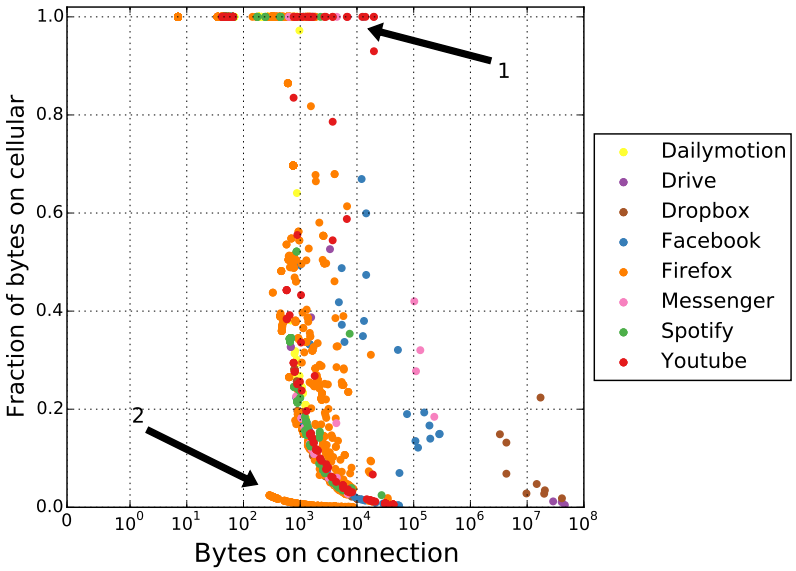


Figure 4.7: When the default route points to the cellular interface, lot of connections are aspired by the WiFi interface.

and 15 Mbps of downstream bandwidth. When the smartphone is attached to both this WiFi access point and the 4G network, the smartphone sends 91% of the data over the 4G network. This allows the smartphone to reach 8-9 Mbits in the upload direction, which is a significant improvement relative to the WiFi bandwidth under shaping. A closer look at the packet traces reveals that Multipath TCP achieves sometimes a lower performance than regular TCP over the 4G network.

As a second test case, we consider our standard WiFi access and the 4G network limited down to a few hundred kilobits per second. This is the shaping enforced by our cellular network once we reach the monthly traffic volume quota. In this case, 98.8% of the bytes are sent over the WiFi interface. This allows Multipath TCP to reach a thoughtput of around 18-20 Mbits, very close to the one achieved by Multipath TCP with 4G and WiFi. Multipath TCP can thus avoid being trapped in a low performance network.

4.7.3 Analysis of the delay seen by applications

Although the distribution of the flow across both interfaces is interesting from a technical point of view, the user has little care for that. Instead, users are interested in two main important characteristics of the network: the throughput and the delay perceived. In the general case, the throughput is not the critical point; rather, most of the smartphone usages require lot of interactivity between the devices and the servers. In order to evaluate the possible benefits of Multipath TCP on this point, we compute the round-trip-time of all data

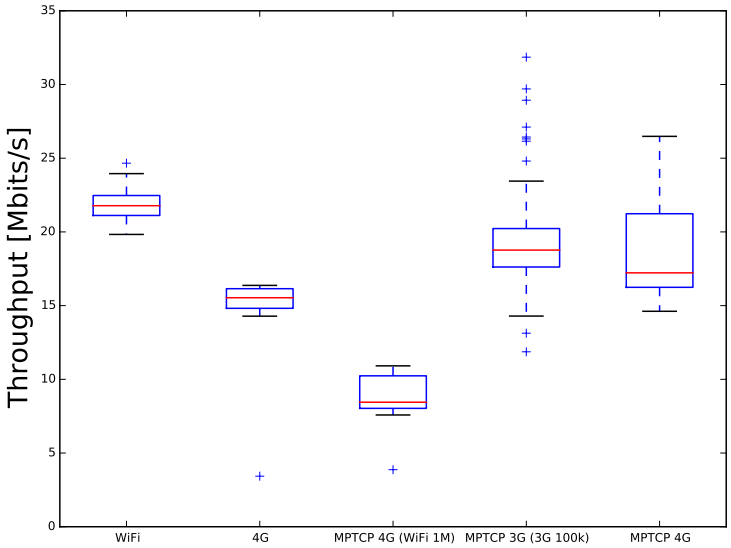
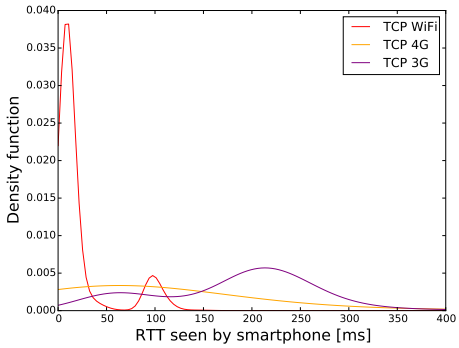
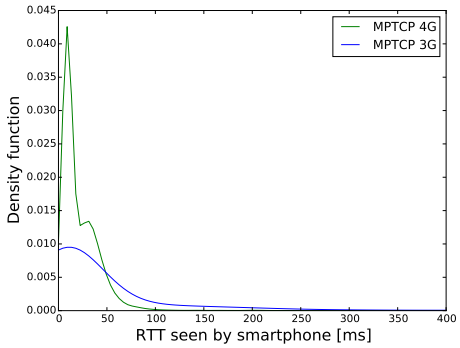


Figure 4.8: Throughput of upload connections with at least 1 MB of data with Dropbox.

packets in the upload direction. The density function function of those round-trip-times is shown on figure 4.9. We only consider traces without shaping in five conditions: TCP with WiFi alone, TCP with 3G alone, TCP with 4G alone, Multipath TCP with both 3G and WiFi and Multipath TCP with both 4G and WiFi. For clarity, we show the results of the three first conditions on figure 4.9a and those of the two last ones on figure 4.9b. Note that for TCP, the round-trip-time is computed by `tcptrace` whereas for Multipath TCP, the round-trip-time is computed at Multipath TCP level by `mptcptrace`.



(a) Using TCP with one interface.



(b) Using Multipath TCP with both interfaces.

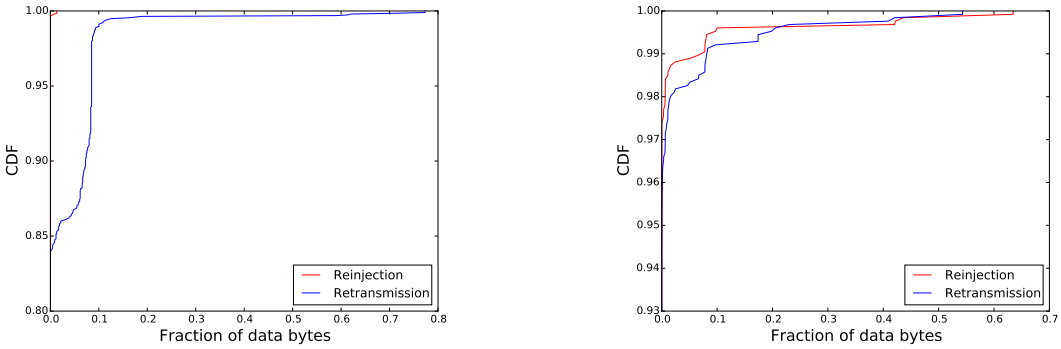
Figure 4.9: Density function of the round-trip-times seen by the smartphone on unshaped networks.

Interestingly, we observe that the density function of the delay indicates a lower delay at Multipath TCP level than the delay obtained by the worst interface alone. Furthermore, in the case of Multipath TCP with both 4G and WiFi, it exhibits lower round-trip-times than

the best interface (which is the WiFi one) and a lower jitter. This indicates that applications will perceive a more interactive and stable network in Multipath TCP than in TCP.

4.7.4 Efficiency of Multipath TCP

In some cases, data transferred by Multipath TCP on one flow can be retransmitted again on the other flow. This phenomenon is called reinjection [63] and might limit the performance of Multipath TCP in some circumstances [68]. We used `mptcptrace` [32] to compute the reinjections over all observed Multipath TCP connections.



(a) From smartphone to proxy.

(b) From proxy to smartphone.

Figure 4.10: CDFs of connections with the fraction of reinjections and retransmissions using MPTCP with both WiFi and 4G up.

In our experiments, reinjections in the upstream direction are rare, as shown on the figure 4.10a. Indeed, fewer than half a percent of all connections include a reinjection, whereas more than 15% of connections encounter TCP retransmissions. Moreover, we observe that those reinjections are short (no more than 5 KB are reinjected on a connection) and only affect biggest connections (no reinjection on connections with less than 1 KB of data). Note that in the figure 4.10, we only consider connections with at least two subflows (since single flow connections are irrelevant in the context of reinjections). In the downstream direction, reinjections are observed on only 2.5% of all connections (as shown on figure 4.10b) but up to 60% of data bytes of a connection can be reinjected. Nevertheless, we have to put those results into perspective knowing that the largest observed reinjection concerned 30 KB on a transfer of 5 MB of data.

4.8 Conclusion

In this chapter, we presented our measurements made using our test framework. This tool is publicly available and can be reused by researchers to conduct reproducible experiments with traffic generated by real applications with a specified scenario.

We studied the interactions between eight very different smartphone applications covering various usages and the latest version of the Multipath TCP implementation in the Linux kernel. First, all the studied applications worked without any modifications with Multipath TCP, showing that the protocol is well compatible with existing applications. Second, for short connections (which are most of the whole connections established by our scenarios), Multipath TCP mainly uses the default route to forward data. This shows the importance of the default route on the smartphone case. A further discussion about that point will be described in chapter 6. Third, for long connections, Multipath TCP enables applications to pool the bandwidth on the cellular and WiFi interfaces and maintains good performances when one of them faces bandwidth restrictions. Since Android smartphones mainly associate to a network by relying on metrics like signal-to-noise ratio, this can be beneficial for the user experience. Fourth, the delay and the jitter perceived by the applications can be lowered by using Multipath TCP in Full Mesh mode, allowing better interactivity and so better user experience. Fifth, the overhead of Multipath TCP related to reinjections is very small compared to the TCP overhead (retransmissions), especially in the upload case.

So far, we discussed about measurements made in controlled scenarios with a motionless device. However, those scenarios are not actual pictures of smartphones usages, but an approximation. Furthermore, users do not stay eternally stuck at a given location. The discussion about the performances of Multipath TCP with traffic generated by real users continues in chapter 5.

Chapter 5

A closer look at real traffic

In the previous chapter, we studied the interactions between Multipath TCP and smartphones applications in a controlled environment. However, although our scenarios cover various kinds of smartphone usages, they give little indications about the traffic generated by real users. The intent of this chapter is to fill this gap by analysing the packet traces generated by two different datasets: a Multipath TCP capable server and Multipath TCP capable smartphones.

There are previous works related to our chapter. Livadariu et al. explore the performance of Multipath TCP on dual-stack hosts in [46] and show that performance over IPv6 and IPv4 paths differ. Ferlin et al. analyse how Multipath TCP reacts to bufferbloat in [28] and propose a mitigation technique. As of this writing, this mitigation technique has not been included in the Linux Multipath TCP implementation. Other measurements have been conducted in the Nornet testbed. Hesmans et al. analyse in [34] a one week-long subset of the `multipath-tcp.org` dataset studied in this chapter. This workshop paper focuses on the control plane aspects of Multipath TCP while our master thesis also considers real smartphones and analyses the packet traces in much more details than [34].

This chapter is organised as follow. First, we describe our two datasets. Then, we propose a broad comparative analysis between our two datasets on various points, like the subflows or the Multipath TCP acknowledgements. After that, we follow a very fine-grained study to explain some imperfections related to the use of Multipath TCP, and in particular to subflows not used, reinjections and receive window limitations. Finally, we conclude this chapter by summarising the key lessons of our measurements.

5.1 Description of the datasets

In this section, we present the two analysed datasets: the `multipath-tcp.org` traces and the smartphones ones. Although the first dataset has nothing particular related to smartphones, it's still interesting to see the behaviour of Multipath TCP in such different environments. As in previous chapter, we still extract information from traces thanks to our analysis scripts explained in section 3.5.1 on page 18.

5.1.1 `multipath-tcp.org` traces

For this first dataset, we use `tcpdump` to collect all packets sent and received by the server that supports multipath-tcp.org. Indeed, the Multipath TCP implementation in the Linux kernel [52] is available from multipath-tcp.org and thousands of users have downloaded and installed it on their computer. In addition to that, this host also supports other web servers, a FTP server and uses an `iperf` daemon to enable researchers and other Multipath TCP users to perform various tests. In the remaining of this chapter, this dataset is called the server dataset. We extract from the traces the packets that correspond to Multipath TCP connections, which can be identified by looking at the utilisation of the Multipath TCP options during the three-way handshake.

Over the five month period from November 17th, 2014 to April 27th, 2015, we have captured more than 400 millions packets from Multipath TCP connections. Overall, 153.5 GBytes of data were exchanged by using Multipath TCP. The server supports both IPv4 and IPv6. Over this period, we received Multipath TCP packets from 7616 different IPv4 addresses (resp. 2496 IPv6 addresses) belonging to 4354 different `/24` subnets (resp. 437 different `/48` IPv6 subnets).

5.1.2 Smartphones traces

Our second dataset covers the traffic produced by a dozen of Nexus 5 smartphones running Android 4.4 with a modified Linux kernel that includes latest Multipath TCP patch (as presented in section 4.5 on page 30). This dataset allows us to understand how real smartphone applications behave when using Multipath TCP instead of regular TCP. However, installing Multipath TCP on the smartphones is not sufficient to use it for all connections established by applications. As of this writing, there are probably only a few dozens of Multipath TCP enabled servers on the Internet and these are rarely accessed by real smartphone applications. To force these applications to use Multipath TCP, we installed `ShadowSocks` on each smartphone and configured it to use a SOCKS-like server that supports Multipath TCP for all TCP connections as already described in section 3.4 on page 15. The smartphones thus use Multipath TCP over their WiFi and cellular interfaces to reach our SOCKS-like server and this server uses regular TCP to interact with the final destinations. The setup is summarised by the figure 3.1 on page 12.

From the server side, all the connections from our dozen smartphones appear as coming from our SOCKS-like server. This implies that the real (cellular or WiFi) IP address of the smartphone is not visible to the servers that it contacts. This might affect the operation of some servers that adapt their behaviour (e.g. the initial congestion window) in function of the client IP address. Note that since `ShadowSocks` does not support IPv6, those traces only contain IPv4 packets. In the remaining of this chapter, we call this dataset the smartphones dataset.

We also installed on each smartphone the MultipathControl Android application (described in section 3.5.2 on page 22) that manages the utilisation of the cellular and WiFi interfaces. Smartphones with Android 4.4 assume that only one wireless interface is active at a time. When such a smartphone switches from cellular to WiFi, it automatically resets all existing TCP connections by using Android specific functions. Our application enables the cellular and WiFi interfaces simultaneously. Our application also controls the routing tables and updates the policy routes that are required for Multipath TCP every time the smartphone connects to a wireless network. Thanks to this application, our modified Nexus 5 can be used by any user since it does not require any networking knowledge.

To collect data, we encouraged our test users to use their smartphones as heavy users. They installed new applications and often listened to web radios when walking. We also used `tcpdump` on the SOCKS-like proxy to collect all the packets viewed by the proxy. In our analyses, we filtered those packets to consider only packets exchanged between the smartphones and the proxy. Without that filter, we could consider Multipath TCP connections between the proxy and a Multipath TCP capable server (like `multipath-tcp.org`), which could bias our results. Over a period of 7 weeks from March 8th, 2015 to April 29th, 2015, we collected more than 65 millions Multipath TCP packets for a total of 20.3 GBytes over 337,587 connections.

5.2 Analysis

We first analyse the main characteristics of the Multipath TCP connections in the two collected datasets. In the server traces, we identify 183,795 Multipath TCP connections. These connections use several destination ports. Most (86.83 percent) connections use port 80. The remaining connections are on ports 21 (4.97 percent), 5001 (1.73 percent) and other port numbers linked to passive-mode FTP connections and a private HTTP proxy used for a few tests. Looking at the exchanged bytes, the connections on port 80 consume 26.8 percent of the total volume (22.14 percent on port 5001 `iperf`).

For the smartphones traces, the destination ports of the captured packets are not sufficient to identify the application level protocol. Since the smartphone connects through a SOCKS-like proxy, all the packets are sent towards the destination port used by our proxy (443 in our case to prevent middlebox interference). To extract the real destination port, we parse the SOCKS command that is sent by the `ShadowSocks` client at the beginning of each TCP

connection and extract the real destination port (details about these messages are given in section 3.4 on page 15). We find that 30.71 percent of the connections use port 53. This is because `ShadowSocks` client sends all DNS requests over TCP to the SOCKS-like proxy. The remaining connections are mainly on ports 80 and 443 with respectively 29.68 and 29.63 percents.

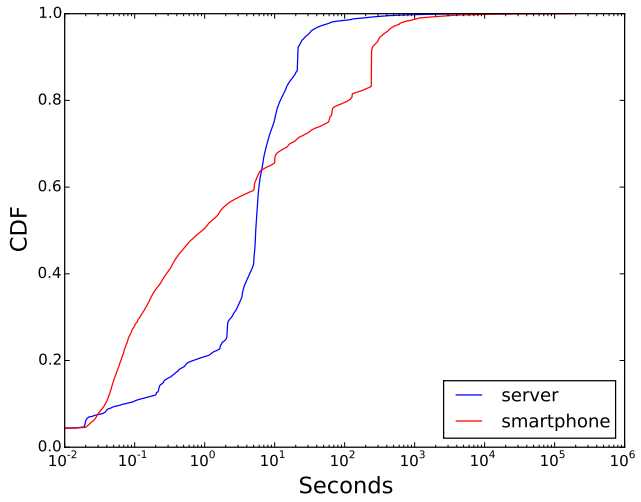


Figure 5.1: Duration of the Multipath TCP connections.

Another interesting point is the duration of the observed connections. Figure 5.1 plots the durations of the Multipath TCP connections observed on both the server and the smartphones datasets. We observe that most of the Multipath TCP connections last less than 10 seconds on both traces: 65.77 percent on the smartphones dataset compared to 75.31 percent on the server one. The smartphones traces contain more shorter and longer connections than the server traces. On the server traces, only 1.51 percent of the connections last more than 100 seconds compare to the smartphones traces with 19.83 percent. Some of these connections on smartphones last for more than one entire day. In particular, we find that one of those one day lasting connections carried only 10 KBytes by establishing (and using) seven different subflows. Looking at the time-sequence diagram, we observe that data is sent in a periodic way with same amount of bytes, except a few packets. This connection is probably related to the smartphone notification services.

Looking at the volume carried by each connection, we observe in figure 5.2 on the next page a different trend than for their durations. On the server traces, we observe that 5.74 percent of the connections carry only two bytes. These correspond to probes that our public server often receives. The two bytes represent start-marker and `DATAFIN` of a Multipath TCP connection. In both datasets, most of the connections carry less than 10 KBytes of data: 86.55 percent on the smartphones dataset and 79.48 percent on the server dataset. In the server dataset, those should be the typical sizes of web connections. But it's important to note that most of the volume is transported in long connections, i.e., connections that last at least several

seconds. As expected, we observe more connections that carry more than one MBytes of data in the server traces than in the smartphones traces.

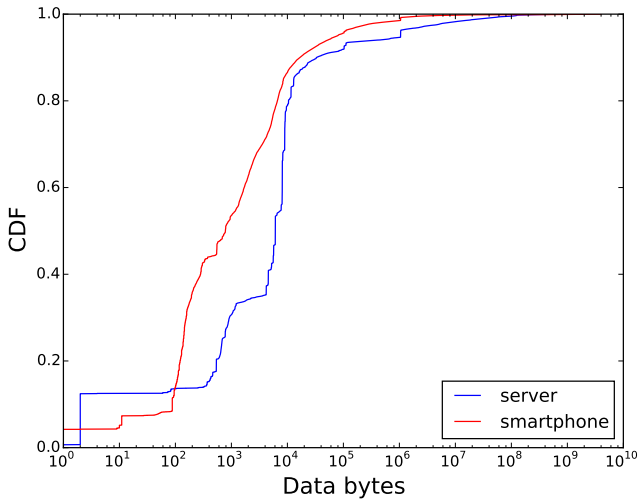


Figure 5.2: Data bytes carried by Multipath TCP connections.

5.2.1 Middlebox interferences

Multipath TCP was designed to cope with a wide range of middleboxes [29, 33]. If the middlebox interferes too much with the operation of Multipath TCP, the connection should fallback to regular TCP. More concretely, if a middlebox modifies the payload, the mapping between Multipath TCP-level and subflow-level sequence numbers is likely to be invalid and this might corrupt the transferred data. Multipath TCP uses its own checksum to detect any modification of payload, ensuring that the mappings are always correct. If the checksum fails, the receiver will inform the problem to the remote peer, close all but one subflow and switch to regular TCP.

Among the 184 thousands Multipath TCP connections in the server traces, we observe only 125 of them falling back to regular TCP, which happened with 28 distinctive client IP addresses. These include 91 HTTP connections and 34 FTP connections. The FTP interference is expected and is due to Application Level Gateways running on NAT boxes. The HTTP interference appears only on the direction from server to client and could have been caused by transparent proxies deployed in cellular and enterprise networks [72]. In the smartphones traces, we do not observe any fallback, but it only covers the cellular networks in a single country that did not have deployed proxies when the traces were collected. The numbers of fallbacks that we observe are lower than expected from earlier work [35].

5.2.2 Establishment of the subflows

With Multipath TCP, a host can send data over different paths (i.e. subflows). The number of subflows that a host creates depends on various factors including the number of interfaces it has and its path-manager as explained in section 2.2.5 on page 7. Table 5.1 reports the number of (not necessarily concurrent) subflows that are observed in our two datasets. For both datasets, we see most of the connections only have one subflow (around 60% for the server dataset and 65% for the smartphones dataset). We analyse in more details the addresses and ports used on the connections that have at least two subflows in the server traces. We find that 49% of these connections were originated from different IP addresses and were likely created by the full-mesh path manager if the remote host was using Linux. The remaining 51% of the connections originated from a single address, which would correspond to the `ndiffports` path manager. All the smartphones that we used were configured with the full-mesh path manager. The number of subflows that they originate depends of the availability of their wireless interfaces, the duration of the connections and the changes of position during the connection lifetimes.

Dataset	1 SF	2 SFs	3 SFs	4 SFs	>4 SFs
Server	61.56%	23.3%	9.17%	1.05%	4.92%
Smartphones	66.28%	31.29%	1.03%	0.53%	0.87%

Table 5.1: Number of subflows per Multipath TCP connection

We observe in both datasets connections that use more than two subflows. In the server dataset, we observe 216 Multipath TCP connections that are composed of 16 subflows. The maximum number of subflows per connection in this dataset is 68. This is not an expected number even if not all subflows are active at the same time since the Multipath TCP implementation in the Linux kernel supports only 32 concurrent subflows. This huge number was probably due to researchers who performed tests with the monitored server. For the smartphones traces, we observe 1773 connections that are composed of 4 subflows (representing 0.53% of all connections) and one connection used 34 different subflows. On the smartphones, having more than two subflows is a sign of handover over different WiFi and/or cellular access points since IPv6 was not used on the smartphones.

5.2.3 Subflows round-trip-times

A subflow is established through a three-way handshake like a TCP connection. Thanks to this exchange, the communicating hosts agree on the sequence numbers, the TCP options that are used and also measure the initial value of the round-trip-time for the subflow. For the Linux implementation of Multipath TCP, the round-trip-time measurement is an important performance metric because as explained in section 2.1 on page 3, the default packet scheduler prefers the subflows having the lowest round-trip-times.

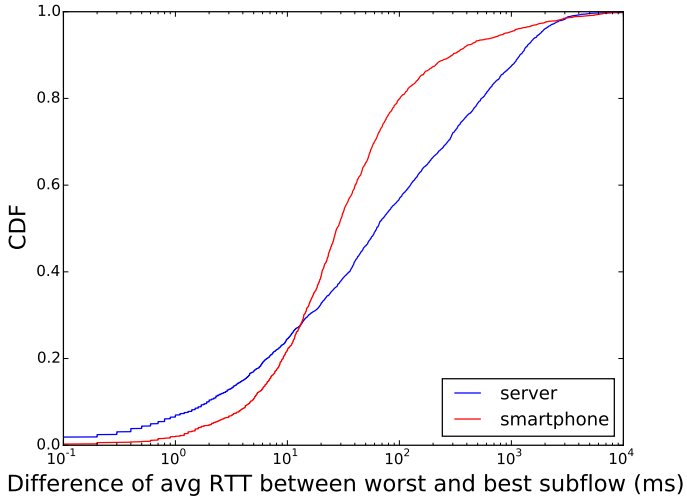


Figure 5.3: Difference of average RTT between worst and best subflows carrying at least 100 KB from server (resp. proxy) to clients (resp. smartphones) on a same Multipath TCP connection.

We evaluate the round-trip-times heterogeneity of the Multipath TCP connections. For this, we first extract from the two datasets the connections that have at least two subflows carrying at least 100 KBytes each. These connections carry most of the data exchanged. For each connection, we use `tcptrace` to compute the average round-trip-time over all the subflows that it contains. Then, we extract for each connection the minimum and the maximum of these average round-trip-times. Figure 5.3 plots the CDF of the delay between the fastest and the slowest subflows over all connections. In the server traces, only 24.6% of the connections have subflows whose round-trip-times are within 10 msec. In the smartphones traces, we observe that 79.9% of the connections are composed of subflows whose round-trip-times are within 100 msec or less. In both datasets, only around 30% of the connections experience a difference lower than 15 ms. 33.5% of connections on the server have a round-trip-times difference larger than 200 msec and 12.6% in the smartphones traces. We observe that bufferbloat is present in both datasets since in 12.5% (resp. 4.6%) of the connections the round-trip-times differ by more than one second in the server (resp. smartphones) traces.

5.2.4 Multipath TCP acknowledgements

As explained in section 2.3.1 on page 8, Multipath TCP uses two levels acknowledgements : the regular TCP acknowledgements at the subflow level and the cumulative Multipath TCP acknowledgements at the connection level. Over the years, TCP has been tuned to pack acknowledgements and minimise their overhead [10]. When there are no loss and no reordering, a TCP receiver should usually acknowledge every second packet that it receives. We clearly observe this expected behaviour in figure 5.4 on the next page that plots the CDF of the number of bytes that are acknowledged by the non-duplicate TCP acknowledgements in the

smartphones trace. We focus on the acknowledgements received by the proxy server since the smartphones do not transmit a lot of data. This plot is a weighted CDF were the contribution of each acknowledgement is weighted by the number of bytes that it acknowledges. We add a filter on Multipath TCP connections with at least two subflows. We observe that 8.15 percent of the TCP acknowledgements were duplicate acks. 11.13% of the TCP acks (accounting for 27.72% of the bytes acknowledgement) acknowledge one entire packet (1428 bytes).

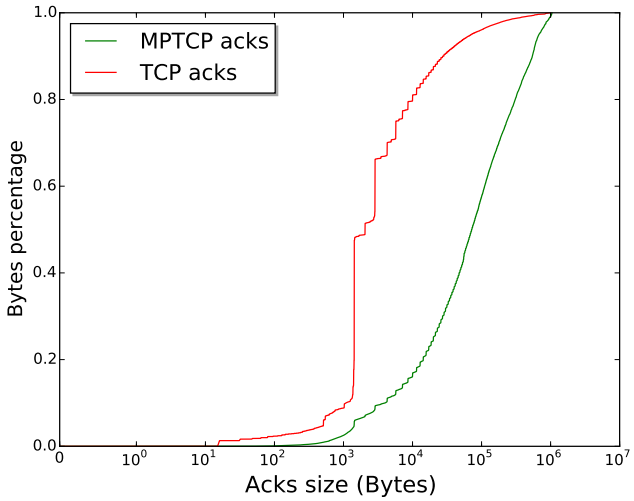


Figure 5.4: Size of the acknowledgements received by the proxy server

We now perform the same analysis by looking at the DSS option that carries the Multipath TCP Data acknowledgements. The green curve in figure 5.4 shows the weighted cumulative distribution of the number of bytes acked per Data acknowledgement. Compared with the regular TCP acknowledgements, the Multipath TCP acknowledgements cover more bytes. In this trace, we observe 12.6% of duplicate acknowledgements at the Multipath TCP level. In the smartphones trace, 16.9% of the Data acks acknowledge 10,000 bytes of data or less while for regular TCP 81% of the acks acknowledge 10,000 bytes or less. Furthermore, 42.2% of the Data acks acknowledge more than 100 KBytes. In the opposite direction (smartphones to proxy, not shown), we do not observe such a huge difference, but this is because the smartphones only send small amounts of data.

The difference between the regular TCP acks and the Data acks is caused by the reordering that occurs when data is sent over several subflows. Since the Data acks are cumulative they can only be updated once all the previous data have been received from all subflows. If a subflow with a long round-trip-time is used, it will cause reordering and data will remain in the reordering queue on the receiver for a long period.

5.2.5 Utilisation of the subflows

The second point that we analyse is the actual distribution of the data among the different active subflows. For this analysis, we select from each dataset the Multipath TCP connections that contain at least two subflows that transmit data. On such a connection, data can be sent in many ways among the different subflows. At one extreme, almost all bytes can be sent over one of the subflows. At the other extreme, packets can be distributed in a round-robin fashion. To evaluate how data is spread among the established subflows, we propose to use the frequency of the *subflow switches*. We compute this number by observing all the packets sent over each Multipath TCP connection. In such traces, we count the number of consecutive packets (based on the DSS option) that are sent over each subflow. The number of *subflow switches* is incremented each time a data packet sent over subflow i is followed by a data packet sent over subflow j, $i \neq j$. For a given Multipath TCP connection, the frequency of the *subflow switches* is then the number of *subflow switches* divided by the duration of the connection in seconds. The advantage of this metrics is that it can be used to compare connections that transfer different amounts of data and have different durations. A high *subflow switches* frequency will indicate a connection that sends the packets in a round-robin fashion among the active subflows. A low frequency will indicate a connection that sends long bursts of data over each subflow.

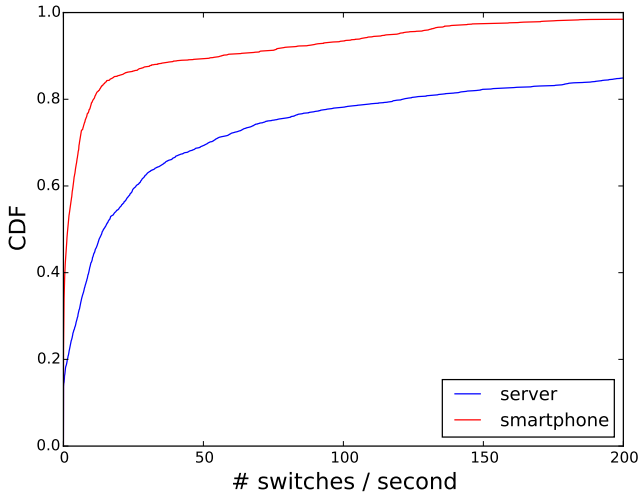


Figure 5.5: Subflow switching frequencies in the two datasets

Figure 5.5 provides the CDF of the subflow switches frequencies computed over all Multipath TCP connections that carry at least 1 MBytes of data with at least two subflows that transmit data. Each point in the curves corresponds to one Multipath TCP connection. In the smartphones traces, 80% of the connections have a subflow switching frequency that is larger than 10. 6.5% of the connections in this dataset switch more than 100 times per second between subflows.

The CDF of the subflow switching frequencies is different in the server traces. In those traces, only 42.5% percent of the connections have a frequency smaller than 10. On the other hand we observe that 15.1% percent of the connections in this trace of have a subflow switching frequency that is larger than 200. The higher subflow switching frequencies in the server can be explained by several factors. First, as explained in section 5.2.2 on page 48, half of the connections in those traces used the `ndiffports` path manager. The subflows on these connections are likely to follow (almost) the same path and thus have (almost) the same round-trip-times. Furthermore, there are probably users that have configured the `round-robin` scheduler and use it to contact the monitored server.

5.3 Multipath TCP imperfections

The previous section has shown how Multipath TCP is used by real applications in today's networks. It confirms the good results obtained by several researchers who performed active measurements with the Linux implementation [54, 12, 18]. However, previous work has shown that there are some network conditions where this implementation does not always perfectly use the available networks [56, 18, 6, 68, 5]. We analyse the packet traces in more details to detect some Multipath TCP imperfections.

5.3.1 Unused subflows

A first potential imperfection Multipath TCP is that subflows can be established without being used to transport data. Creating subflows that are not used consumes some bytes and more energy on smartphones [57] since the interface over which these subflows are established is kept active. Considering Multipath TCP connections composed of at least two subflows in the smartphones dataset, we observe that among the 249,622 subflows established, 135,136 of them were additional subflows (i.e., not initial ones). 102,287 of these subflows were established without carrying any data.

There are three reasons that explain those unused subflows. Firstly, the subflow can become active after all the data has been exchanged. This happens often since 53.6% of the connections in the smartphones traces carry less than 1000 bytes of data. Secondly, the difference in round-trip-times between the two subflows can be so large that the subflow with the highest RTT is never selected by the packet scheduler. Usually, the cellular subflow has a larger RTT than the WiFi subflow. Though, we notice that on the unused additional subflows, 43.15 percent show a better RTT with the newly-established subflow, but 75 percent of the connections containing such subflows carry less than 1000 bytes, and 95 percent less than 20 KBytes. Thirdly, the subflow can be established as a backup subflow. For the server dataset, we observe 66,287 subflows that did not carry any data. By immediately creating subflows, the `full-mesh` path manager is responsible for some Multipath TCP inefficiencies. An improved path manager [68, 57] would prevent some of these inefficiencies.

5.3.2 Reinjections

A second possible source of imperfections with Multipath TCP are the reinjections. A *reinjection* [63], is the transmission of the same data over two or more subflows. A reinjection can occur for several reasons: *(i)* handover, *(ii)* excessive losses over one subflow or *(iii)* limited windows due to the Opportunistic Retransmission and Penalization (ORP) heuristic proposed in [63] and enhanced in [56]. This phenomenon has been shown to limit the performance of Multipath TCP in some wireless networks in [68].

Among all the analysed Multipath TCP connections, only 2.5-3% of them contain reinjections in the server dataset. This percentage increases up to 8-9% in the smartphones dataset. However, if we focus on the connections that are composed of at least two subflows, these numbers increase. In the server (resp. smartphones) dataset, 4.1 (resp. 10.6) percent of these connections experience reinjections.

Multipath TCP reinjections are closely coupled with regular TCP retransmissions. We use `mptcptrace` (resp. `tcptrace`) to extract all the reinjections (resp. retransmissions) in our two datasets. Since reinjections can only occur on connections that contain at least two subflows by definition, we perform this analysis by considering only the connections that are composed of at least two subflows and that carry at least one byte. Figure 5.6 shows the CDF of the reinjections and retransmissions in the server dataset. The number of retransmitted and reinjected bytes are normalised with the number of bytes exchanged over the connection. Since the same data can be sent over several subflows, this fraction can be larger than one for connections that carry a small amount of data that is retransmitted and/or reinjected.

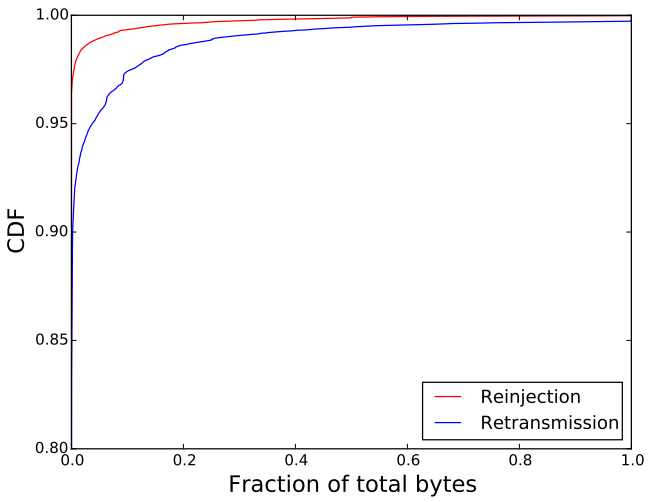


Figure 5.6: CDF of the fraction of bytes that are reinjected/retransmitted on the Multipath TCP connections composed of at least two subflows in the server dataset

We observe that reinjections occur but they are less frequent than the regular TCP retrans-

missions. While 25% of the Multipath TCP connections composed of a least two subflows experience retransmissions, only 4.1% of them experience reinjections. For the server dataset, 923 MBytes (718,327 packets) are reinjected (out of a total of 113 GBytes transmitted). In those traces, 1323 MBytes are retransmitted.

To better understand the reinjections and the retransmissions, we plot in figure 5.7 the CDF of the normalised times when reinjections and retransmissions occur during each Multipath TCP connection. To produce this plot, we extract from the packet traces the timestamps of all retransmissions and reinjections and normalise them as a fraction of the duration of the entire Multipath TCP connection. Note that here, we include all connections (and thus also single-flow ones), which allow to see all retransmissions (and not only on multi-flow connections). Moreover, each point corresponds to the timestamp of one retransmitted/reinjected packet. In the server trace, we observe that there are proportionally more retransmissions in the beginning of connections than close to the end. This is probably because TCP's congestion control algorithm is more aggressive during the slow-start phase than when it enters congestion avoidance mode. For the reinjections, we do not observe in figure 5.7 a strong bias in favor of the beginning of the connections.

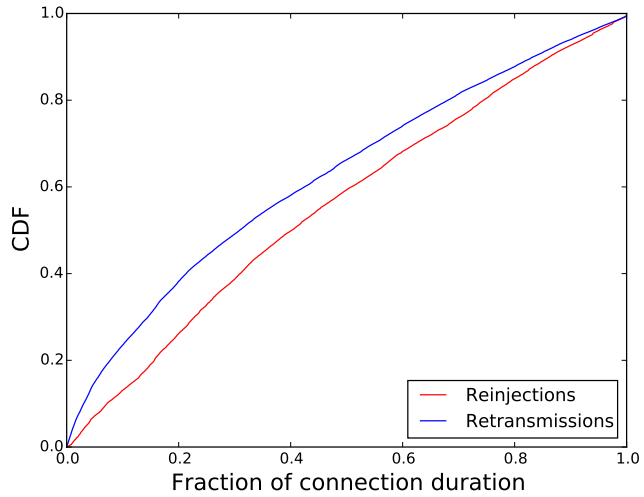


Figure 5.7: CDF of the normalised times when reinjections and retransmissions occur in the server dataset.

Figure 5.8 on the facing page also provides the CDF of the fractions of the bytes that are retransmitted/reinjected in the smartphones traces. We observe that these two events are more frequent than in the server trace. 27.7% of the connections composed of a least two subflows experience at least one retransmission. Furthermore, 50% of the bytes sent over a connection are retransmitted in 2.3% of the connections. Looking at the reinjections, we observe the same trends. There are more reinjections in the smartphones trace than in the server trace. 10.6% of the connections experience reinjections and 2.3% of them reinject 20% or more of their bytes. However, looking at the total number of bytes, the problem is not so

severe. In the smartphones dataset, we observe only 63.7 MBytes (resp. 237 MBytes) of data that are reinjected (resp. retransmitted) by the proxy to the smartphones. This number must be compared with the 19.8 GBytes of data that are sent by the proxy to the smartphones in this dataset.

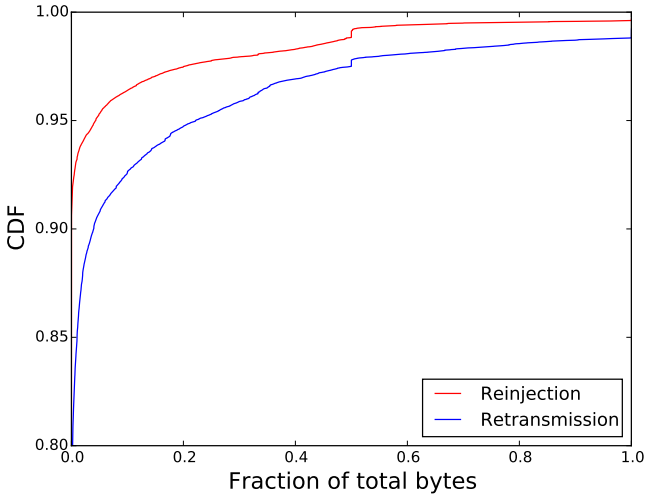


Figure 5.8: CDF of the fraction of bytes that are reinjected/retransmitted on the Multipath TCP connections composed of at least two subflows in the smartphones dataset.

The behaviour of retransmissions and reinjections is quite different when looking at the smartphones dataset, as shown on the figure 5.9 on the next page. Indeed, retransmissions occur much more at the beginning of the connections (more than 35% of the retransmissions occur during the first fifth of the connections) and at the end (more than 15% of the retransmissions occur close to the very end of the connection). This behaviour is driven by two reasons. First, smartphones use different wireless interfaces: cellular (2G, 3G, 4G) and WiFi. Second, some connections on smartphones (e.g. HTTP/1.1) tend to carry most of their data bytes at the beginning of the connection and then the connection remains almost idle until its end.

For reinjections, we observe several vertical bars in figure 5.9 on the following page. A closer look at the packet traces reveals their root cause. The bar at the fraction 0.33-0.34 is caused by a quite long connection (lasting more than 30 minutes) that performs a handover. Five subflows were established for this connection. It exchanges most of its 450 MBytes of data within the first seven minutes. At that point, a window of more than 115 KBytes of data (carried with very small packets, such as these represent more than 6000 packets) over around 3 seconds was reinjected. In this connection, the main cause of the reinjection is the loss of a cellular subflow, probably caused by a user movement. After trying to retransmit some packets over the cellular subflow, Multipath TCP reinjects them over another subflow.

The vertical bar at 0.95 on figure 5.9 on the next page is also mainly due to one connection that performs handover. This connection lasted 800 seconds and transferred more than

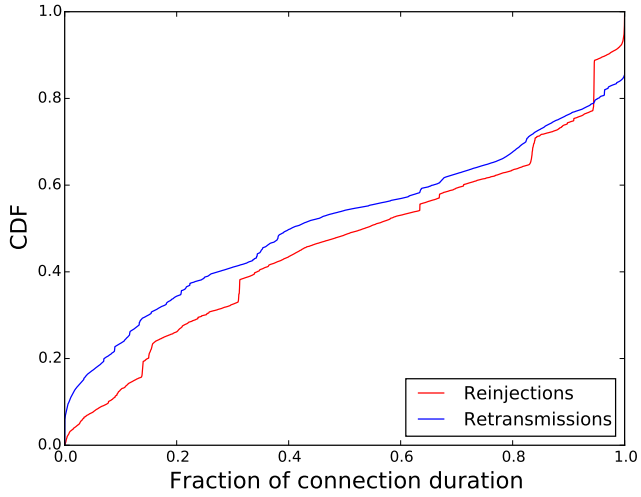


Figure 5.9: CDF of the normalised times when reinjections and retransmissions occur in the smartphones dataset.

4 MBytes of data over three subflows. At the beginning of the connection, the smartphone used one subflow, and the server sent data slowly (around 100 KBytes over 5 seconds). This subflow then failed when only one packet was in flight. During a period of eight minutes, the server tried to retransmit the packet without success. Then, the smartphone opened a new subflow, the server retransmitted the segment and then started to send data at a higher rate (200 KBytes within one second). After one second, this subflow failed while 175 KBytes of data were still in flight. Three minutes later, the smartphone opened a third subflow and the server sent one reinjected data packet to the smartphone that acknowledged it. However, the server waited one more minute before reinjecting the data bytes that were in-flight (with some retransmissions). Overall, 14,531 packets were reinjected on this connection. On this last subflow, 3.5 MBytes of data were sent within 50 seconds.

The reinjections that appear nearly at the end of connections are mainly caused by the same kind of events as described before. When analysing the retransmissions that appear close to the end of connections we observed than 42.1% occurred on short connections lasting less than 3 seconds.

5.3.3 Receive window limitations

The receive buffer plays a more important role for Multipath TCP than for regular TCP since it is used to cope both with retransmissions but also to reorder the data received over the different subflows. With single-path TCP, if the receive window is too small, then the sender will slow down. Multipath TCP uses a single window that is shared among all active subflows [63]. If the receive window is too small, the sender is blocked. This blocking might be

caused by the transmission of some data over a slow subflow. To cope with such limitations, Multipath TCP can use the Opportunistic Retransmission and Penalisation (ORP) heuristic to reinject this data over another subflow.

To evaluate whether the receive window caused performance problems in our traces, we first extract the maximum receive window advertised over each Multipath TCP connection. Figure 5.10 provides a boxplot of these maximum windows in function of the number of bytes transported over each connection of the server dataset. For the short connections, we observe a huge variance among the maximum advertised receive windows with a median of 92 KBytes. For the connections that carry more data, the median maximum advertised window increases with the connection size. This variation is expected since the automatic buffer tuning algorithm [66] used by Linux automatically adjusts the receive window during the connection. For the long data transfers, the receive window appears to be large enough in most cases. In the smartphones traces, all smartphones are configured with the same settings for the receive buffer.

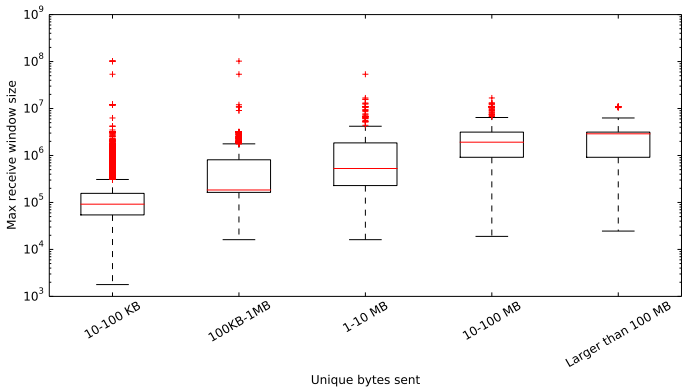


Figure 5.10: Maximum advertised receive windows in the server traces

Despite the large maximum receive buffers that are advertised in our traces, we still observe reinjections that are triggered by the ORP heuristic due to a limited receive window. To detect those ORP reinjections, we use `mptcptrace` to extract the number of in-flight bytes and compare it with the advertised receive window for each transmitted packet. Our detection works in two steps. We first extract from the packet traces the timestamps when the difference between the advertised receive window and the flight-size is too small. We consider this difference to be too small if it is below $T_{WinDiff}$ that we set at a value of 5,000 bytes. Then, if a reinjection occurs within 50 milliseconds after such a timestamp we consider that it is caused by the ORP heuristic. Among the connections that are composed of at least two active subflows, we detect that 33.7% of the reinjections are caused by the ORP heuristic.

We analysed many of these connections manually with the graphs produced by `mptcptrace`. As an example of the good operation of the automatic buffer tuning, let us consider the figure 5.11 on the next page showing the following connection extracted from the server

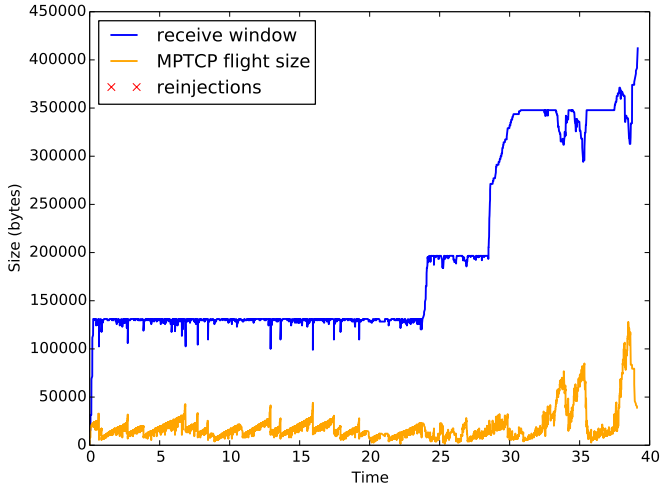


Figure 5.11: An example of the automatic buffer tuning working as expected.

dataset. At the beginning of the connection, the receive window grew to reach a first limit at about 125 KBytes. After around 25 seconds, the receive window increased, showing the establishment of a new subflow for the connection. During the whole connection, the number of bytes in flight never reached the receive window, showing that the sender was not blocked by the receive window. However, our qualitative analysis reveals some unexpected interactions between this heuristic and the receive window.

In some cases, we observe issues with automatic receive buffer tuning that caused unnecessary early reinjections. In those cases, the received buffer announced by the receiver does not grow fast enough to cope with the multiple flows. This too small receive buffer blocks the sender that decides to reinject data. It is important to note that from the subflow TCP perspective, the sender is not blocked by the window because it does not take into account data sent over the other subflows in its own window. An example that illustrates this particular case is presented in figure 5.12 on the facing page. The red crosses at the top of the graph show the times when reinjections occur. We observe that there are early reinjections at the beginning of the connection while the receive window has not yet been fully opened by the automatic buffer tuning algorithm. Once the receive window has reached its maximum value, there are no more reinjections. These reinjections cause a temporary slowdown of the connection that may be relatively important for small connections because a good subflow may be penalised for a bad reason.

Another factor that may limit the performance of Multipath TCP is the size of the receive window when the difference between the round-trip-times of the subflows is too large and the fastest subflow is not sufficient to carry all the data. In this case, Multipath TCP tries to use several subflows. If the current receive window is too small, Multipath TCP will reinject data in an attempt to unlock the situation faster. Note however that the reinjected segment must

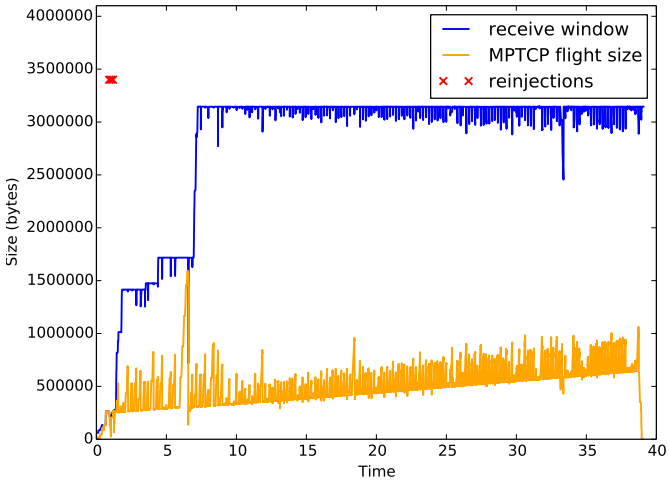


Figure 5.12: Receive buffer autotuning leading to unnecessary reinjections

still fit in the congestion window of the subflow on which it reinjects the data. Unfortunately, there is no guarantee that the reinjected segment will arrive faster than the original one. Figure 5.13 on the next page is an example extracted from our packet traces that illustrates this situation. In this case, reinjections are caused by a window limitation and happen on regular basis during the entire connection lifetime. A more detailed analysis of the Multipath TCP time sequence graph reveals that the reordering is mainly responsible for this usage of the receive window. To prevent this problem, the packet scheduler on the sender should be replaced by a scheduler that sends segments out-of-order so that they would arrive in order at the receiver. Such schedulers have been proposed and evaluated by simulations [42], but to our knowledge they have not been included in a real Multipath TCP implementation.

5.4 Conclusion

In this chapter, we have shed some light on how Multipath TCP is used today by real applications in both fixed and wireless networks. We expect that our findings and the datasets that we have collected will help Multipath TCP implementers and other researchers to improve the protocol and its implementations so that it can reach the same stability and efficiency that TCP reached after decades of usage.

Our detailed analysis of two different kinds of packet traces has led to several interesting observations. First, the server traces shows that Multipath TCP works correctly over a wide range of Internet paths. This demonstrates the deployability of the protocol extension in the global Internet. Second, the server and the smartphones traces reveal two different usages of Multipath TCP. About 70% of the connections are composed of subflows that are created

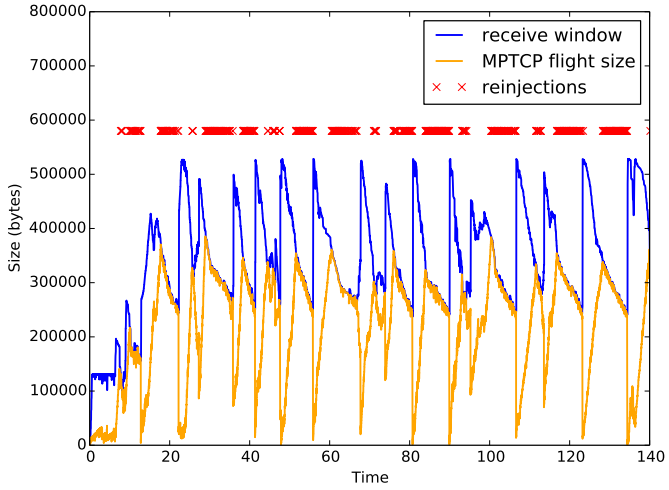


Figure 5.13: Frequent reinjections on a window limited connection

almost immediately. These connections expect improved performance from the utilisation of several subflows. In the remaining 30% of the connections, one or more subflows are created more than one second after the initial subflow. This corresponds to the second use case for Multipath TCP that enables applications to continue to use existing connections after a handover. This use case is very important for mobile devices. Looking at the round-trip-times of the subflows that Multipath TCP uses, our analysis reveals that there can be a huge difference between the fastest and the slowest subflow inside each Multipath TCP connection. This large delay difference must be taken into account by Multipath TCP developers who propose solutions to improve the performance of the protocol. To analyse how data is spread among the active subflows, we have introduced the subflow switching frequency. This metric measures how often data transmission switches from one subflow to another.

We have then analysed in more details some of the inefficiencies of Multipath TCP. The first identified inefficiency is that the current Multipath TCP implementation in the Linux kernel often creates subflows that do not carry any data. On a server, this is not a severe problem, but on a smartphone, establishing a subflow on the cellular interface without using it has a cost in terms of energy consumption and radio channel usage. Multipath TCP developers should consider better path management strategies for smartphones. Second, we have studied in details how retransmissions and reinjections occur. Our analysis reveals that reinjections, i.e. retransmitting data over more than one subflow, are common but less frequent than regular retransmissions. However, when they are triggered to react to receive window limitations, they can significantly affect the connection performance. The ORP heuristic that is currently used by the Linux Multipath TCP is probably not the perfect approach to solve this problem.

Chapter 6

Streaming applications

Our third measurement campaign is about streaming applications and more precisely about radio streaming applications. This kind of application is particular because it generally transfers a constant flow of data over long connections. Thanks to Multipath TCP, these connections can remain alive while switching from one wireless network to another. These handovers are frequent with smartphones as we are often disconnected from a WiFi network while moving out of building. Because users are also attached to a cellular network, they expect their current connections to continue to work on this cellular network without interruption. This behaviour is not natively supported by TCP but Multipath TCP has been designed to handle such cases as previously explained in chapter 2 on page 3.

The study of mobile devices using with Multipath TCP is not new. Paasch et al. provide in [54] detailed measurements on the interactions between Multipath TCP and cellular and WiFi networks. They proposed and analysed three modes for the operation of Multipath TCP. In particular, they consider reaction to handovers. Raiciu et al. analysed in [62] a mobility architecture that uses Multipath TCP on mobile nodes that access servers through proxies. Their measurements performed with Multipath TCP on a laptop in movements demonstrate the benefits of such an architecture. Pluntke et al. [59] analysed whether Multipath TCP could reduce energy consumption by using several interfaces simultaneously. Lim et al. proposed in [45] a technique to reduce the energy consumption of smartphones using Multipath TCP and evaluated it with experimental measurements. Xiao et al. [77] analysed the performance of regular TCP in mobility scenarios. They identified that one of the main performance issues was the handoffs between networks in which TCP is disconnected for some time. Here, we can see that Multipath TCP could resolve this issue by opening a new subflow for every arising network, instead of waiting the loss of a TCP connection to

open a new one. Williams et al. analyse in [73] the performance of Multipath TCP in moving vehicles. They observed that at moderate speed, Multipath TCP is able to remain active and perform handover without adversely affecting a delay-tolerate file download or causing excessive data-level retransmissions.

The problematic related to handovers has also been analysed in other studies. Croitoru et al. [15] explored a WiFi-based solution that allows clients to continuously scan for nearby access points and add a new virtual network interface each time a new router is discovered. Nirjon et al. [49] proposed a way to perform handovers based on TCP connections. However, this last solution assumes that the TCP connections are quite short, and is not suitable to long ones such as the streaming ones.

In this chapter, we first introduce our methodology used to interpret collected data. We then analyse cases in which the applications try to support handovers themselves using regular TCP in a mobile environment. The benefits of Multipath TCP and its backup mode are then analysed. Finally, we conclude this chapter by giving some areas of improvements for Multipath TCP.

6.1 Methodology

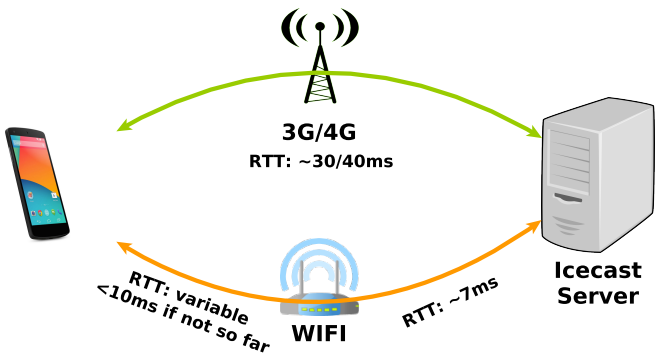


Figure 6.1: Schema of the infrastructure for these tests with RTT values.

We installed Icecast [78] services combined to the Music Player Daemon [20] (`MPD`) in order to stream a constant audio stream to the smartphones. To make a slight digression and according to [36], streaming a constant flow is more energy-consuming than only sending bursts of data. Even if this mechanism is not efficient on smartphones, Icecast is a popular streaming media server used by many public radios in the world. By setting our own Icecast services on our Multipath TCP-ready server, we were able to easily dump packets generated by this service and fully control the application and its environment. But more importantly, no proxy is required to perform the measurements. We thus obtain results which really show what we could have if Multipath TCP is supported by streaming media providers.

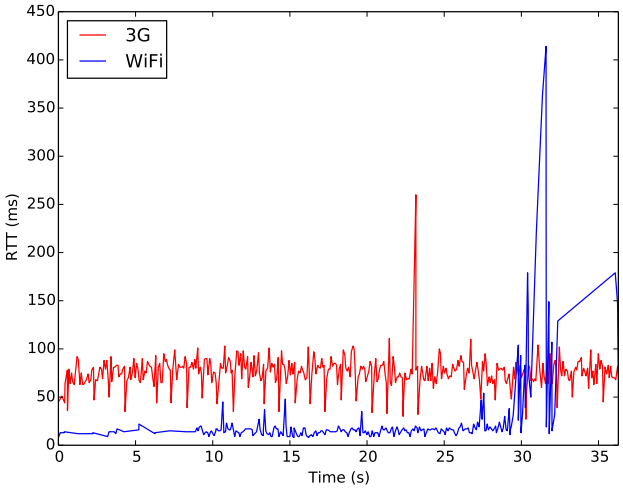


Figure 6.2: Variation of the RTT on both interfaces when the devices moves away from a WiFi access point

Five WiFi routers connected to the 100 Mbps campus network have been deployed around our work area. Round-trip-time mean values are visible in figure 6.1 on the preceding page. The RTT between the smartphone and the WiFi routers is very variable when walking around but when they are close, the RTT between the smartphone and the server is always lower than the cellular network. An example of the variation of the round-trip-times caused by the growing distance between the device and a WiFi access point is shown on figure 6.2. These access points do not cover the entire area, some parts are only covered by the cellular network. This corresponds to a scenario where a user walks in a city where some homes share a public WiFi access point (e.g. FON). WiFi with WPA2 encryption was enabled on all these devices but without any WPA Enterprise feature, routers are not linked together and there is no roaming at all. By deploying these routers, we wanted to increase the number of handovers when moving smartphones along them.

During our tests, two Nexus 5 devices were used simultaneously: an unmodified one using regular TCP and one supporting Multipath TCP with our MultipathControl application described in section 3.5.2 on page 22. The unmodified smartphone uses the Cubic TCP congestion control algorithm but we tested several coupled congestion control schemes with the other Nexus 5. A dedicated Icecast daemon per smartphone was launched on our server in order to avoid that Icecast prioritises one over the other. The sound emitted by the streaming application on the smartphones was recorded with the Shou.tv Android application[1]. With these audio files, we can generate a visual representation of the sound produced by the TuneIn radio[2] streaming client application. We also performed some tests with the Free and Open

[1] See: https://shou.tv
[2] See: http://www.tunein.com

Source application ServeStream[3] but it seems that this application doesn't automatically restart streaming audio files if the smartphone switches from a WiFi to a cellular network. We guess these developers didn't want to generate unwanted cost by using the cellular network without user's agreement but we wanted to avoid manual controls on handover.

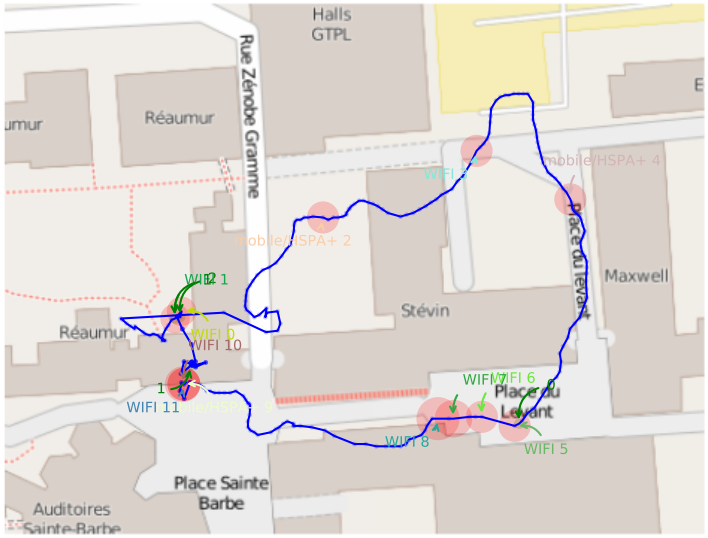


Figure 6.3: A map with a subset of information recorded by MultipathControl application.

We tried to walk along the same path during approximately five minutes for all these tests. An example is displayed on the figure 6.3. On this map we can see the journey in blue, when and where our MultipathControl application was notified of a network change in red with an annotation about the network type and its ID. The accuracy of the path shown on this map depends on the GPS chip.

Data from traces are extracted with our scripts described in section 3.5.1 on page 18 such as `seq_together.py`. Other scripts specially made to analyse data generated by our MultipathControl application are available in [7].

6.2 Overview

An overview of the situation with and without Multipath TCP is discussed here.

6.2.1 Handover at application level

According to [48], TuneIn Radio supports handover at the application level. This is confirmed by figure 6.4 on the facing page. It shows the sound produced by this application when turning

[3] See: http://sourceforge.net/projects/servestream

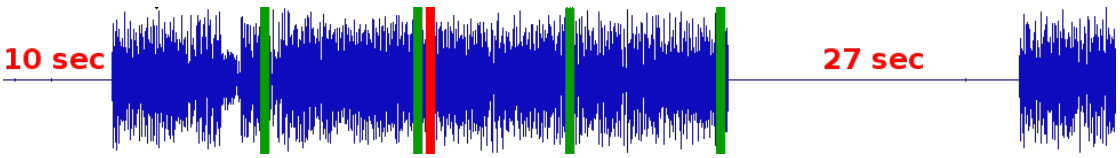


Figure 6.4: Sound produced by TuneIn Radio when turning on an off the WiFi interface every 15 seconds on the unmodified smartphone.

on and off the WiFi interface every 15 seconds after having played the first sound. These events are marked by the green lines on the figure. We can see that the application is still able to play sound from its buffer. Sometimes the user will not notice these switches because most media stream servers (including Icecast) are configured to send a burst of data at the beginning of the download in order to quickly fill in a buffer. Looking at the traces, we can see that TuneIn Radio starts downloading a new stream via the new default interface. It sends this request to the server without any special option which would let it receive data from a specific timestamps (e.g. no `HTTP` range option). Furthermore, it receives no indication from the server to help it supporting handovers. Then we guess that the application drops the content that has already been downloaded from its previous request and puts the new one at the end of its buffer. But these operations are not always perfect because we sometimes hear tiny audio glitches that are almost non perceptible and marked by a red line on the figure. During our tests we also noticed that some server configurations force the application to manage handovers by skipping the non-received music and playing the only current live one sent by the server. Parts of the music were skipped but on the other hand, this application still continued to produce sound.

Supporting handovers at the application level is not an easy task. It cannot be performed with all protocols and can have some limitations as seen on this figure 6.4. After a few network changes, we can see that the application has stopped producing sound (horizontal line in the figure 6.4) as too much content have been dropped during handovers at the application level. It also takes a lot of time to restart playing stream media. This kind of situation could be even worse when the device starts to be far from a still attached WiFi access point.

6.2.2 Mobility

Supporting handovers when walking is harder because the conditions vary depending on the device's position compared to the access point. When walking, we cannot say that we have only two states: a perfect connection or no one. But when moving the device away from the access point, the signal quality gradually gets worse. It means that when being far from an access point, a lot of TCP retransmissions occur and the throughput decreases below the streaming rate. This is confirmed by figure 6.5 on the next page. The blue (resp. red) curve shows the quantity of bytes transferred during a walk (see figure 6.3) via the WiFi (resp. cellular 3G) interface. Vertical blue and red bars represent moments where no more data was transferred on the specified interface. Green ones indicate when TuneIn Radio's handover

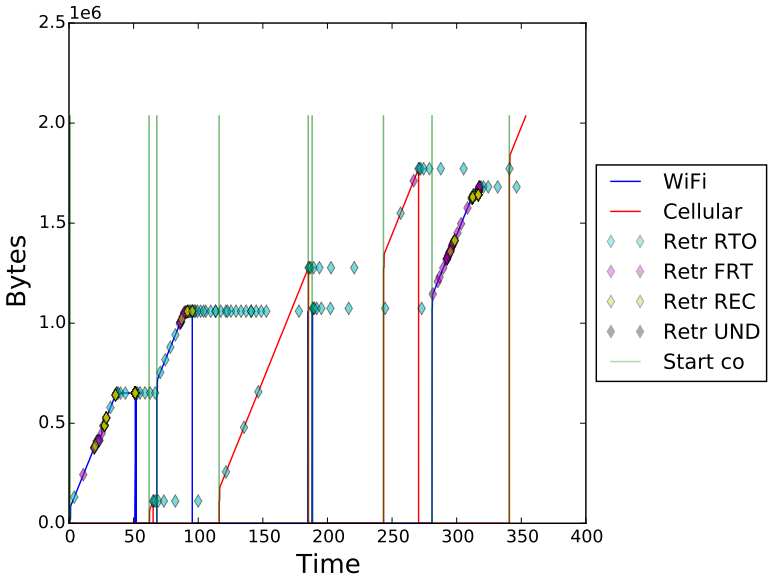


Figure 6.5: Quantity of bytes transferred during a walk with the unmodified smartphone.

mechanism is triggered by asking the server for a new HTTP `GET` request in order to download streaming content. Finally this figure also indicates when different retransmissions packets have been seen.

Four kinds of retransmissions are detected: retransmissions due to a timeout (`RTO`), fast retransmits (`FRT`), recover (`RCO`) and unneeded ones (`UND`). (*i*) When they are due to a timeout, packets are re-sent after a variable time period. This timer value depends on different factors: it has to be larger than the RTT but has also to take into account at least the size of the receiving windows and how many times the timer has expired for the same packet. (*ii*) Fast retransmits are sent either after two or three duplicate ACKs as defined by [3], due to SACK [47] information or resulting of the congestion control algorithms, i.e. Reno's Fast Recovery algorithm [67]. (*iii*) The third case of retransmissions are the recover ones. They happen when fast retransmission packets are sent a second time. (*iv*) Retransmissions can be seen as unneeded if a retransmit is sent too quickly or for a wrong recover. It is important to keep in mind than these retransmissions are sent and detected by the server. It is this possible that not all these retransmission packets have been received by the smartphone.

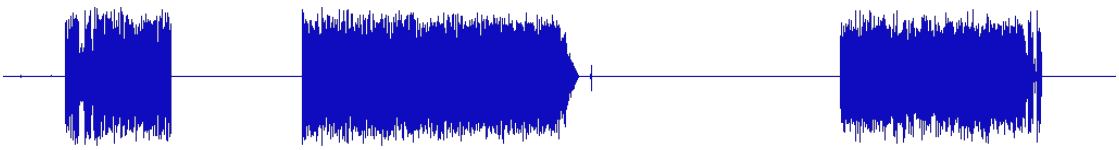


Figure 6.6: Wave form played by TuneIn Radio during the same walk as in figure 6.5 with the unmodified smartphone.

When looking at the retransmissions visible on the figure 6.5 on the facing page, we can see that most of them occur just before a wireless interface becomes unavailable. It is a sign that the smartphone is loosing packets and the WiFi connection will soon go down. On the other hand we see that new download requests, represented by green vertical bars, are not immediately sent by the application when no more data is transferred to the Smartphone's WiFi interface but could take from 10 to 55 seconds, the longest time we saw during our tests. These long delays explain that TuneIn Radio's buffer will quickly be emptied and will not be refreshed by the new stream content received via the new default interface. On wireless networks, it is not easy to know exactly when the network is unreachable because a device can only rely on timeouts: this value is difficult to estimate as if this period is too short, the device will stop using this interface whereas other transient events can also cause delays. If it is too long, then all applications which need network access will be blocked instead of restarting their requests via the backup interface. Because it takes too much time to be notified when a disconnection from an unreachable WiFi access point happens, the user will notice that the streaming application is not able to play any sound during few seconds as shown on figure 6.6 on the preceding page. This figure shows the wave form played by TuneIn radio during the same walk and the same unmodified smartphone used to generate the previous figure 6.5 on the facing page.

Even if these figures 6.5 and 6.6 are generated from the same walk, we did not align them. To do so, we would have to extract data from the application about when it starts downloading data compared to the moment when it starts playing music; how much data are dropped when trying to support handover at application level; what is the size of the buffer when loosing connections, etc. Since this application is not Open Source, we cannot easily analyse it to extract these information. On the other hand, the goal of these figures is to show that the application is not able to support handovers correctly. We don't want to try to predict the exact moment when the application will stop producing sound because it depends on unknown and variable conditions. But we can see that if the application stops receiving any data during a few seconds, it will no longer be able to continue providing streaming services.

6.2.3 Streaming over Multipath TCP

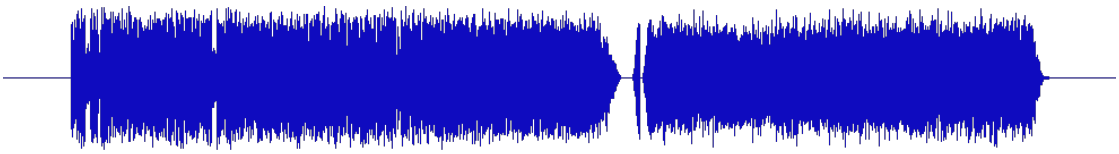


Figure 6.7: Wave form played by TuneIn Radio during a walk when using Multipath TCP.

When looking at the figure 6.7 representing the sound produced by the streaming application on the smartphone using Multipath TCP for the same walk, we can see that the application continues to produce the sound without any blank. Compared to figure 6.6 on the facing page where regular TCP was used, the improvement is significant. There is no more interruption in

the song played by the application which means that handovers are much better supported. Note that the small blank at around the middle of the wave is actually caused by a change of music: fade out of music A, fade in of music B.

6.3 Analysis

We now analyse in details the collected traces. In particular, we check how the traffic is balanced between the WiFi and the cellular interfaces and we discuss the efficiency of the backup mode.

6.3.1 Traffic distribution

It is interesting to analyse the distribution of the traffic among the cellular and WiFi interfaces because the user might prefer the WiFi interface over the cellular one. The main reason is the cost in terms of money but according to [45], also in term of consumed power energy. Depending of the network type, the available bandwidth can be higher but the round-trip-time is often lower when using WiFi as seen in chapter 4.

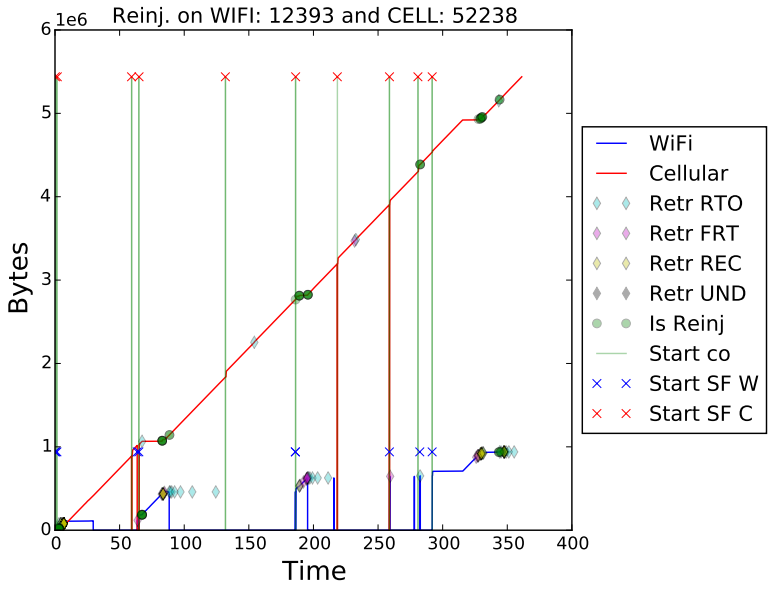


Figure 6.8: Quantity of bytes transferred during a walk when using Multipath TCP with LIA, WiFi interface as default route and 3G.

Figure 6.8 shows that most of the traffic is sent over the cellular interface. The default Multipath TCP scheduler is used. It prefers the subflow with the lowest RTT and favours the one with fewer retransmissions [55]. It is exactly what we see on this figure 6.8: when both

interfaces are attached to a network, WiFi is firstly used when a new request is performed by the client — also because the default route is via the WiFi interface as explained in section 4.7.1 on page 34. Then when the first reinjections occur, the cellular interface takes the reins until the next new request. According to the green vertical bars on the figure, it seems that the application is still notified when a change in the network arises and then starts a new subflow. The WiFi interface is then used at the beginning of this request which receives a burst of data. But after a few seconds, all the traffic is sent through the cellular network and some packets are even reinjected to this second subflow for a total of 50 kB.

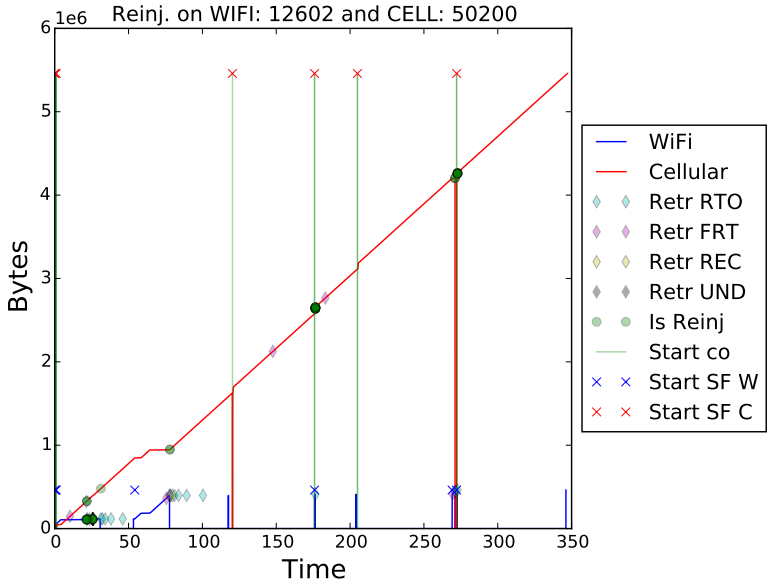


Figure 6.9: Quantity of bytes transferred during a walk when using Multipath TCP with LIA, cellular interface as default route and 3G.

When the cellular interface is used as default route and as visible on figure 6.9, almost all traffic is sent through the cellular network. Compared to the previous situation, half of the traffic is sent over the WiFi network but the reinjected bytes are almost the same. Note that here, new Multipath TCP subflows are created because either the application creates new connections or the smartphone is connected to different access points along the walk. It means that the counters are started from zero for each new subflow and that is why Multipath TCP's scheduler tries to send data through the WiFi network which could have a lowest RTT.

As observed in chapter 4, if the smartphone is connected to the same router and the cellular network is preferred by Multipath TCP's scheduler due to some TCP retransmissions, most of the traffic will go through the cellular network. This is mainly explained because the reception and transmission windows are not full at all. As shown on the different figures, Multipath TCP has then no need to use more than one subflow to transfer data at a low rate and

this is why there is no reason to switch to another subflow, even more if this subflow is less stable. Furthermore, current Multipath TCP implementation has no probing mechanism. In this situation where the kernel has no need to use several subflows, it will not try to resend data on the subflow which had transitional problems even if the RTT is now lower. It means that Multipath TCP will only have an overview of a previous state.

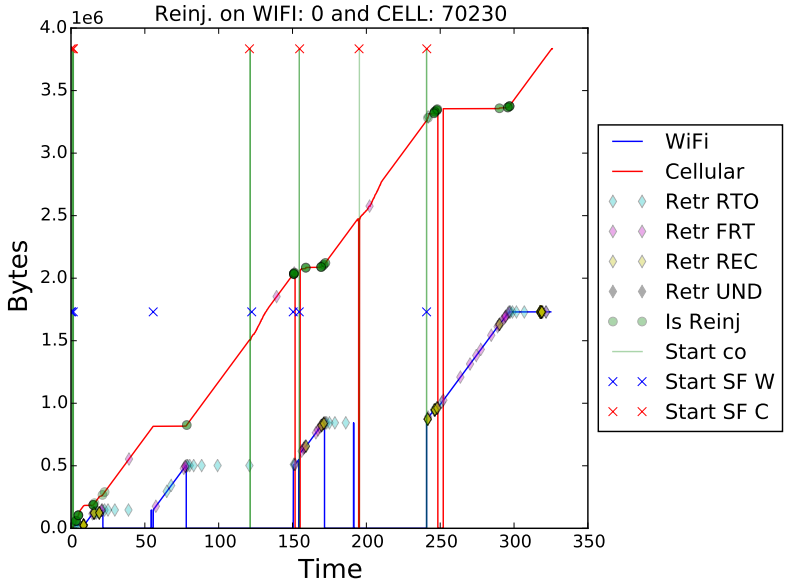


Figure 6.10: Quantity of bytes transferred during a walk when using Multipath TCP with LIA, WiFi interface as default route and 2G.

When the 2G technology is used, handovers are still well supported: the stream application continues to play sound without any blank. Figure 6.10 shows that this 2G network has just enough bandwidth to support the traffic generated by the Icecast server. Bursts of data take a few more time to be received but the round-trip-time is much more important: RTT seems to be around 200 ms which is 6 times more than with 3G/4G. We can see that the WiFi network is now much more used, even if multiple retransmissions — mostly categorised as fast retransmit — are detected on this interface.

Other coupled TCP congestion schemes have been tested in addition to LIA. All results are quite the same as presented here with LIA when using either OLIA [39] or wVegas [11]. This can be explained by the fact that the application does not pull data as fast a possible from the server. Therefore, the TCP received window of the lowest round-trip-time path is not fully filled, thus only one subflow is used. Seeing no difference between those coupled congestion control algorithms is thus expected.

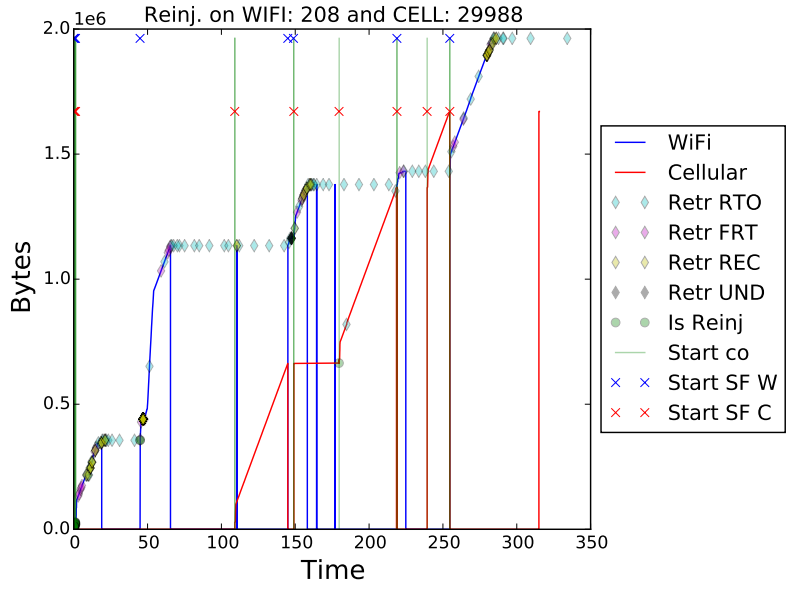


Figure 6.11: Quantity of bytes transferred during a walk when using Multipath TCP with LIA, WiFi interface as default route, cellular as MPTCP backup link and 3G.

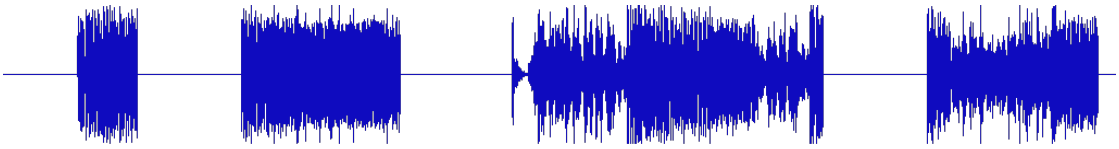


Figure 6.12: Wave form played by TuneIn radio during the experiment of figure 6.11.

6.3.2 Multipath TCP's backup mode

Multipath TCP supports a backup option [29] that was described in section 2.2.4 on page 7. When adding it to the subflows established over the cellular interface, the traffic on this cellular network should be reduced while still taking advantage of Multipath TCP's handover mechanism. But the current implementation of this backup option seems not aggressive enough to support handovers in our case.

Indeed, figure 6.11 shows that fewer data is sent through the cellular network. But we can also see that Multipath TCP only uses this backup subflow after several retransmissions due to a timeout. As result, the application is not able to fill its buffer and cannot play any sound when it is empty. This is confirmed by the wave form on figure 6.12.

We are facing here the same problem as the one with the handover supported at the application level. The good point is that if the device switches from one to another WiFi access point, a new subflow is created and all data are recovered without any losses. But if this switch is not fast enough like the one at the beginning of the experiment in figure 6.11 (around

50 seconds), the buffer is emptied and no sound is played as shown in figure 6.12 on the preceding page (the first interruption).

The server is not immediately notified when the smartphone is too far from the access point or disconnected. We can even see on figure 6.11 on the previous page that the cellular interface is only used when a new download is requested while only this interface is available. The situation is equivalent to supporting handover at the application level because in both cases the cellular network is only used when the connection on the WiFi interface is lost.

6.4 Conclusion

The positive point is that Multipath TCP and its different congestion algorithms support well handovers. In other words it means that the user is able to listen to music while walking without having any interruption even if the application tries or does not try to support handover itself.

On the other hand and after analysis, we can see that the cellular network absorbs most of the traffic mostly because it is more stable and always available. But this side effect is maybe not wanted by the user because using the cellular network is often more expensive. The WiFi interface should be more frequently used, a bit like what we have when using 2G technology where WiFi is used until no enough data can be sent to the smartphone via this interface.

These results show that Multipath TCP should be customised for this case. Applications could also be adapted. For instance they could give information to the server to prefer using WiFi network over cellular except if the server is not able to send data at sufficient rate. It implies that Multipath TCP has to be customised, e.g. via SOCKET's options. Multipath TCP's scheduler and path manager have also to be adapted.

About Multipath TCP's backup option, it should also be improved according to these results. Multipath TCP's scheduler should not wait too much before using backup subflows if there are too many retransmissions on the non-backup ones. Data could be sent over the backup subflow when a certain amount of losses and retransmissions have been detected. The client should maybe take the decision to allow using the backup subflow. As a workaround, the client could send a `MP_PRIO` packet to notify the server that a subflow is no longer a backup one. To favour one subflow over the other one, this scheduler could also try to estimate the waiting time in the send-buffer queue. If data is waiting there more than a defined time, these bytes could then be sent over the other subflows. All these changes could improve user's experience with smartphones applications and especially the streaming ones.

Chapter 7

Conclusion

To sum up this Master's thesis, the main important point is that Multipath TCP works well on smartphones without any modification at the application level. Results show that smartphones applications are now able to increase the bandwidth utilisation by using multiple interfaces at the same time. Moreover, Multipath TCP supports keeping connections alive along different networks, which is something difficult or even impossible at application level. Alongside different predefined scenarios, we also conducted real medium-scale ones in which we observed very long connections on smartphones, during more than one day.

With Multipath TCP's default RTT-based scheduler and its Full Mesh path manager, we observe that the smartphone balances the network traffic between its WiFi and cellular interface for some connections. Those must last at least two round-trip-times on the additional interface and the application must push data faster than the growth of the congestion window. For instance, this occurs when the application tries to push data as quickly as possible upon the establishment of the connection. In that way, connections are not stuck by a low performance network; they use the other one to compensate. However, this behaviour is maybe not what the user always wants, since there are additional factors, such as the monetary cost of each device or the power consumption, that should be taken into account.

Nevertheless, our measurements reveal that an important fraction of the connections are quite short, i.e. shorter than one second. For those, Multipath TCP may not be useful, and may even involve additional overhead without any benefits. In addition, some issues related to the current reinjection strategies were found. This shows the need of an adaptation of Multipath TCP to the particular case of the smartphones.

7.1 Future work

To our knowledge, the analysis of the behaviour of Multipath TCP on smartphones with real traffic was an unexplored area of research. Our results have then open several lines of thoughts for future works; some of them are proposed here.

As a result of the study, the native handovers support and bandwidth utilisation on mobile could be easily improved thanks to Multipath TCP. This promising extension could then be already put it in place. On the mobile side, this could be done by integrating it in the upstream kernel instead of using a customised Multipath TCP-ready kernel. On the server side, this could be conducted towards the main Internet actors to increase the number of end servers supporting Multipath TCP. In the meantime, the mobile providers could significantly improve their own slower end-user network by already supporting it in their intermediate infrastructure. Some of them already use transparent proxies (also called middleboxes) to optimise their mobile network. Adding Multipath TCP support on these servers could already let their clients benefit of the Multipath TCP advantages.

Regarding the Multipath TCP extension itself, a first step would be to improve the integration of Multipath TCP for the smartphones environment. Related to our different chapter conclusions, the path manager could be modified to open additional subflows after some time or some transferred bytes in order to lower the number of unused subflows for short connections. Another option could be to develop a better scheduler to meet the expectations of the user, depending of its needs, such as monetary cost or saving energy. About the integration in Android devices, the default behaviour after being connected to a WiFi access point is to disable the cellular interface. With our MultipathControl application, we counter this effect, but the application could not be always quick enough to react, leading to the closing of subflows established on that interface. To avoid this effect, a better integration with the Android framework could be implemented as discussed in section 3.5.2.

The section 2.3.2 introduced the congestion control algorithms. However, we didn't study in details the effect of the available algorithms on the performances of Multipath TCP on a mobile environment. In particular, it could be interesting to observe how those algorithms react when a data heavy connection is spread across both interfaces (e.g. watching high quality videos).

It could also be interesting to perform measurements with SPDY/HTTP 2 protocol which seems to be the future of HTTP 1. In particular, we could see how Multipath TCP reacts to those longer connections compared to HTTP 1(.1).

Except in chapter 6 with the streaming measurements, a SOCKS-like proxy was used between the smartphone and the servers to allow the smartphone to use Multipath TCP for all TCP connections. Although this proxy slightly affects the observed behaviour (see for instance the section 4.7.1 and the behaviour of Firefox), most of remote servers are not Multipath TCP capable yet. With the agreement of several big actors of the Internet or providers to support Multipath TCP on test servers, similar measurements should be conducted again with direct

connections between the smartphone and the servers.

As detailed in appendix C, we faced issues when trying to perform measurements on IPv6. Indeed, all our results with smartphone traffic were obtained on IPv4. It might be interesting to replay those measurement campaigns on IPv6 to see if it has an impact or not on our conclusions.

A last point to look at when studying mobile devices is their consumption. Indeed, algorithms running on such devices must be energy efficient to prevent them from draining all the battery. In particular, the current full mesh path manager opens subflows on both interfaces and thanks to our MultipathControl application, we let them on. The impact of such decision on the energy consumption should be studied.

Appendix A

Tools used

A.1 tcptrace

tcptrace [50] is a tool written by Shawn Ostermann to analyze TCP traces. Initially started in 1994, the tool has not majorly evolved since 2004 and the last version, 6.6.7. It takes a PCAP file as input and produces graphs to characterise TCP connections in addition to various global statistics. This is the base tool that we used to obtain information on TCP traces. However, tcptrace does not understand specific options and messages of Multipath TCP, and so cannot be used alone to analyse Multipath TCP traces. Moreover, running it on Multipath TCP traces sometimes gives strange results, in particular with the number of data bytes. Notice that some bugs are open on the Ubuntu bug tracker of tcptrace[1]. That's why our analysis scripts check if values are coherent before using them.

[1] https://launchpad.net/ubuntu/+source/tcptrace/+bugs

A.2 mptcptrace

mptcptrace [32], developed by Benjamin Hesmans, is an analyser of Multipath TCP traces. It is publicly available[2]. During this master thesis, we slightly improved this tool to give additional information and fixed a few bugs. Notice that the version on the master branch on the Git repository was designed to process only quite short traces, which is not compatible with big traces collected and analysed in chapter 5. The dvlp_branch branch brings important improvements to cope with large files, and we used this version to produce our results.

Unlike tcptrace, mptcptrace understands the basics of the Multipath TCP protocol and can extract the keys that are exchanged during the establishment of the initial subflow from the MP_CAPABLE option. Thanks to these keys, mptcptrace computes the tokens that identify the Multipath TCP connection on both the client and the server. With these keys, mptcptrace can link the different subflows that compose each Multipath TCP connection.

Once the subflows that are associated to one Multipath TCP connection have been grouped together, mptcptrace is able to produce metrics and graphs to analyse it. In contrast with tcptrace, mptcptrace mainly uses the contents of the DSS option instead of the regular sequence number and acknowledgement number present in the TCP header. Among graphs, it can plot the throughput (instantaneous, over X packets, on average) at the Multipath TCP level, the size of the receive window or the amount of bytes that are in flight (i.e., not yet acknowledged). Like tcptrace, it can also produce global statistics, at the difference these are computed at Multipath TCP level. For instance, the duration of a Multipath TCP connection is defined as the time between the first SYN that carries the MP_CAPABLE option and the last packet sent over this connection, but not necessarily on the same subflow. Another important element detected by mptcptrace is the reinjection, i.e. the transmission of the same data over two or more subflows, and is important to characterise the efficiency of Multipath TCP.

[2] See https://bitbucket.org/bhesmans/mptcptrace

A.3 tcpdump

tcpdump [70] is a tool that dumps network traffic into a PCAP file. This format is readable by our scripts, but also by various other tools like Wireshark. The code and the development are publicly available[3]. It offers various filtering options, such as the interface to listen to our expressions packets need to match. We used this tool to capture packets either on a specific virtual interface on our server and on the smartphone.

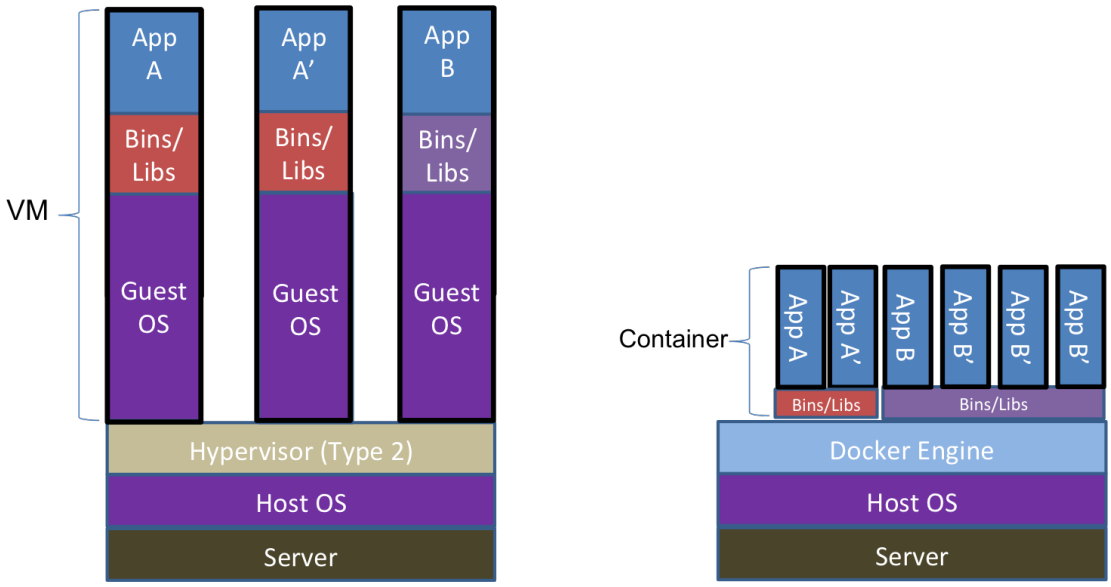
A.4 tcpcsm

tcpcsm [2] is a tool initially developed in 2011 at the University of Waikato. Its purpose is to detect TCP congestion behaviours. It can detect loss or reordering events, track congestion window size and understanding why TCP sends less data than expected, using packet header traces as input. By specifying an option, we can do additional analysis of traces with our analysis scripts (described in section 3.5.1 on page 18) with this tool. In our case, we use it to track the origin (retransmission timeout, fast retransmit,...) of TCP retransmissions in chapter 6.

[3] See http://tcpdump.org

A.5 Linux containers with Docker

On our dedicated proxy, we had to launch different services. It was interesting to launch them on isolated environments for technical and security reasons. Indeed, a weak service will not impact other services, we can control each environment separately, we can use virtual interfaces and only dump traffic dedicated to one service, plus many other positive points.



(a) For each isolated service, an entire OS with libraries and programs has to be virtualised.

(b) For each isolated service, only applications are launched on isolated processes and some resources can be shared between containers

Figure A.1: Comparison of Virtual Machines and Containers. Source: Docker.com

Virtual Machines (VM) virtualise an entire system in isolated areas as shown in figure A.1a. Using VM is particularly interesting when we have to test different kernel versions because virtualisers like qEmu[4] allow us to virtualise a system and indicate a path to a custom kernel image. It's then easy to switch from one to another kernel image. Furthermore we received some scripts to setup new environments, etc. Unfortunately, our Kimsufi server doesn't support hardware acceleration to virtualise systems. Without it, the processor gave us very bad performances, it was not usable.

Because we didn't need to regularly change the kernel, it was then decided to use Linux containers with the help of Docker[5]. Linux containers are considered as operating-system-level virtualisators and allow multiple isolated user space instances. It still use the same kernel space as the host machine but applications can be launched in isolated environment as shown

[4] See http://qemu.org
[5] See https://docker.com

in the figure A.1b on page IV.

It was interesting to use Docker to quickly create, run, debug and duplicate services. All public services were running on a separated container. For each container, a dedicated virtual network interface is created and connected to a virtual ethernet bridge named `docker0`. Docker allows us to easily create `iptables`'s rules to redirect all traffic with a specific destination port to one precise container. Then it's easily possible to create one environment (e.g. a SOCKS proxy) and launch multiple instances with the same configuration but accepting connections with different destination ports on the host machine.

Appendix B

Running services on server

For this thesis, we created and modified a few `Dockerfiles` which looks like Bash scripts and are used to build new Docker images. All of them are available on our Github account[1]. Here is the list of services that we used, which are only free and open source ones.

B.1 Proxy services

- `OpenVPN`: We tried to use a VPN (Virtual Private Network) as proxy to use Multipath TCP for all TCP connections but it was too heavy.
  https://openvpn.net
- `SSH` tunnel: A SSH SOCKS proxy can be quickly set up. A dedicated service allowing only the use of a SOCKS proxy was set up. This solution was finally abandoned due to the overhead of using SSH encryption and mostly because this created a tunnel where all TCP connections were piped into a single one. Having one single connections was not easy to analyse and not realistic.
  http://sshtunnel.googlecode.com
- `ShadowSocks`: This proxy is based on SOCKS protocol [44]. We decided to use the implementation in `C` because it's a lightweight service with a very small footprint. More details about this selected proxy are given in the section 3.4 on page 15.
  http://shadowsocks.org

[1] See https://github.com/MPTCP-smartphone-thesis

B.2 Sharing services

- `FDroid`: This service analyses all Android packages (APK) of a selected directory and generate XML and images files needed to serve a `FDroid` repository with a HTTP server. This repository of Android applications can be used with the `FDroid` client, originally created to propose a catalogue of Free and Open Source Software. This one was used to easily propose update for our Android applications.
  https://f-droid.org
- `NGinx`: This is a popular web server and it was needed to distribute files of our `FDroid` repository.
  http://nginx.org
- `SSH` access: SSH access to some files has been granted to either collaborators and our machine controlling our smartphone dedicated to our tests.
  http://www.openssh.org
- `Pure-FTPd`: An FTP server was also used to share some files.
  http://pureftpd.org

B.3 Custom services

- `Collect MPCtrl`: To collect data sent by our MultipathControl application described in section 3.5.2 on page 22, a simple HTTP/RESTful server has been created. It simply listens to HTTP PUT messages with some `JSON` content and add data in a MongoDB database.
  https://github.com/MPTCP-smartphone-thesis/server-collect-mpctrl
- `MongoDB`: A MongoDB service was running on a container linked to the previous one. With Docker we can easily restrict access to selected containers.
  https://mongodb.org

B.4 Streaming services

- `Icecast 2`: A streaming media server. It has been use to server MP3 files during the last phase of our researches.
  http://icecast.org
- Music Player Daemon (`MPD`): This tool was used to send a constant MP3 flow to the Icecast server. The MP3 files come from public Jamendo radios.
  http://www.musicpd.org

B.5 Miscellaneous services

- `Collectd` and `Graphite`: `Collectd` was installed on the host machine to collect information about the utilisation of the CPU, discs, networks, etc. All data were transferred to a container and displayed with `Graphite` web application.
  https://collectd.org – https://launchpad.net/graphite
- `pdnsd`: a proxy DNS server. Because both `Shadowsocks` clients and servers don't support IPv6 as explained in section 3.4 on page 15, we set up a modified version of this DNS proxy in order to not answer with IPv6 results. But due to a lack of time, we prefer to disable IPv6 on Android devices given to our testers.
  http://members.home.nl/p.a.rombouts/pdnsd

Appendix C

IPv6 and smartphones

Because ShadowSocks does not support IPv6[1], we didn't want that the smartphone tries to first looking for IPv6 addresses before IPv4 ones if the device has a global IPv6 address. Indeed, all IPv6 traffic is not redirected to the proxy and it is not what we want. Furthermore, we faced some errors where IPv6 addresses were unreachable on some networks. It was not really a problem for web browsers which implements Happy Eyeballs [74] algorithm. But other applications could first try to connect to a IPv6 address before falling back to a IPv4 address only after a few seconds.

Different approaches can help us fixing these issues with IPv6:

- Applications should always prefer IPv4 over IPv6. It's not possible to modify the code of all applications but we could change the behaviour of `getaddrinfo()` function to get an IP from a domain name. Unfortunately, Android doesn't use GNU C Library (glibc) which allows some configurations[2]. To avoid having to recompile Android's `libc` and re-install on all smartphones, we prefer to find another solution.
- It's possible to block DNS requests with IPv6 addresses with `pdnsd` settings as proposed on the Github issue[1] but we needed to manually reinstall and reconfigure ShadowSocks client on all clients.
- Redirecting DNS traffic to a local proxy DNS server on the proxy and modify the code to only answer with IPv6 addresses could be a solution. But a new extra layer for all

[1] See https://github.com/shadowsocks/shadowsocks-android/issues/275: IPv6 is not blocked in China then it's not an urgent task according to the developers.

[2] See http://man7.org/linux/man-pages/man5/gai.conf.5.html

DNS requests will be added and we don't want to slow them more.

- Disabling IPv6 on the smartphones could be seen as a radical fix but it's an easy fix and on the other hand, IPv6 should not be used. It's possible to disable IPv6 by recompiling the Android kernel without its support but to avoid reinstalling a new version on all devices, it was decided to simply change a `sysctl` setting. Notice that Android resets this setting when a interface is re-enabled, which is managed by our MultipathControl application presented in section 3.5.2 on page 22. Then a `iptables`'s rule has been added to block IPv6 traffic and then avoid receiving a global IPv6 address.

Bibliography

[1] Alcock, S., and Nelson, R. Application flow control in youtube video streams. *SIGCOMM Comput. Commun. Rev. 41*, 2 (Apr. 2011), 24–30.

[2] Alcock, S., and Nelson, R. Passive detection of tcp congestion events. In *Telecommunications (ICT), 2011 18th International Conference on* (2011), IEEE, pp. 499–504.

[3] Allman, M., Paxson, V., and Stevens, W. Rfc 2581: Tcp congestion control.

[4] Alvarez, C. Network monitor. https://github.com/caarmen/network-monitor/, 2015.

[5] Arzani, B., Gurney, A., Cheng, S., Guerin, R., and Loo, B. T. Deconstructing mptcp performance. In *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on* (Oct 2014), pp. 269–274.

[6] Arzani, B., Gurney, A., Cheng, S., Guerin, R., and Loo, B. T. Impact of path characteristics and scheduling policies on MPTCP performance. In *Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on* (May 2014), pp. 743–748.

[7] Baerts, M., and De Coninck, Q. Streaming analysis scripts. Available at https://github.com/MPTCP-smartphone-thesis/streaming-scripts.

[8] Baerts, M., and De Coninck, Q. Test framework: Android's ui tests and python scripts. Available at https://github.com/MPTCP-smartphone-thesis/uitests.

[9] Baerts, M., and Detal, G. Multipathcontrol, gui to manage mptcp settings and log statistics on android devices. Available at https://github.com/MPTCP-smartphone-thesis/MultipathControl.

[10] Blanton, E., Duke, M., Braden, R., Eddy, W., and Zimmermann, A. A roadmap for Transmission Control Protocol (TCP) specification documents. *RFC7414* (2015).

[11] Cao, Y., Xu, M., and Fu, X. Delay-based congestion control for Multipath TCP. In *Network Protocols (ICNP), 2012 20th IEEE International Conference on* (Oct 2012), pp. 1–10.

[12] Chen, Y.-C., Lim, Y.-s., Gibbens, R. J., Nahum, E. M., Khalili, R., and Towsley, D. A measurement-based study of MultiPath TCP performance over wireless networks. In *Proceedings of the 2013 Conference on Internet Measurement Conference* (New York, NY, USA, 2013), IMC '13, ACM, pp. 455–468.

[13] Choffnes, D., and Govindan, R. Investigating transparent web proxies in cellular networks. In *Passive and Active Measurement: 16th International Conference, PAM 2015, New York, NY, USA, March 19-20, 2015, Proceedings* (2015), vol. 8995, Springer, p. 262.

[14] Cisco. Vni mobile forecast highlights, 2015. See http://www.cisco.com/c/dam/assets/sol/sp/vni/forecast_highlights_mobile/index.html.

[15] Croitoru, A., Niculescu, D., and Raiciu, C. Towards wifi mobility without fast handover. Usenix NSDI.

[16] De Coninck, Q., and Baerts, M. Analysis scripts. Available at https://github.com/MPTCP-smartphone-thesis/pcap-measurement.

[17] De Coninck, Q., and Baerts, M. Http/rest server to collect data from multipathcontrol's android application. Available at https://github.com/MPTCP-smartphone-thesis/server-collect-mpctrl.

[18] Deng, S., Netravali, R., Sivaraman, A., and Balakrishnan, H. WiFi, LTE, or both?: Measuring multi-homed wireless internet performance. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (New York, NY, USA, 2014), IMC '14, ACM, pp. 181–194.

[19] Detal, G., Paasch, C., and Bonaventure, O. Multipath in the Middle(Box). In *HotMiddlebox '13* (2013), pp. 1–6.

[20] developers, M. Music player daemon (MPD). http://www.musicpd.org, 2015.

[21] Developers, N. Numpy, 2015. See http://www.numpy.org/.

[22] Drago, I., Mellia, M., M. Munafo, M., Sperotto, A., Sadre, R., and Pras, A. Inside dropbox: Understanding personal cloud storage services. In *IMC '12* (2012).

[23] Eardley, P. Survey of MPTCP Implementations. Internet-Draft draft-eardley-mptcp-implementations-survey-02, IETF Secretariat, July 2013.

[24] Eddy, W. M. Defenses against tcp syn flooding attacks. *The Internet Protocol Journal 9*, 4 (2006), 2–16.

[25] Evdokimov, L. redsocks. http://darkk.net.ru/redsocks, 2013.

[26] Fall, K. R., and Stevens, W. R. *TCP/IP illustrated, volume 1: The protocols*. addison-Wesley, 2011.

[27] Ferlin, S., Dreibholz, T., and Alay, Ö. Multi-path transport over heterogeneous wireless networks: Does it really pay off? In *Proceedings of the IEEE GLOBECOM* (Austin, Texas/U.S.A., December 2014), IEEE.

[28] Ferlin-Oliveira, S., Dreibholz, T., and Alay, O. Tackling the challenge of bufferbloat in multi-path transport over heterogeneous wireless networks. In *Quality of*

Service (IWQoS), 2014 IEEE 22nd International Symposium of (May 2014), pp. 123–128.

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

[30] Gember, A., Anand, A., and Akella, A. A comparative study of handheld and non-handheld traffic in campus wi-fi networks. In *PAM'11* (2011), pp. 173–183.

[31] Google Inc. Monkeyrunner. Available at http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[32] Hesmans, B., and Bonaventure, O. Tracing Multipath TCP connections. *SIGCOMM Comput. Commun. Rev. 44*, 4 (Aug. 2014), 361–362.

[33] Hesmans, B., Duchene, F., Paasch, C., Detal, G., and Bonaventure, O. Are TCP extensions middlebox-proof? In *Proceedings of the 2013 Workshop on Hot Topics in Middleboxes and Network Function Virtualization* (New York, NY, USA, 2013), HotMiddlebox '13, ACM, pp. 37–42.

[34] Hesmans, B., Tran-Viet, H., Sadre, R., and Bonaventure, O. A first look at real Multipath TCP traffic. In *Traffic Monitoring and Analysis*, M. Steiner, P. Barlet-Ros, and O. Bonaventure, Eds., vol. 9053 of *Lecture Notes in Computer Science*. Springer International Publishing, 2015, pp. 233–246.

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

[36] Hoque, M. A., Siekkinen, M., and Nurminen, J. K. Energy efficient multimedia streaming to mobile devices—a survey. *Communications Surveys & Tutorials, IEEE 16*, 1 (2014), 579–597.

[37] Hunter, J., Dale, D., Firing, E., Droettboom, M., and the matplotlib development team. matplotlib, 2015. See http://matplotlib.org/.

[38] Kelley, A. Waveform. https://github.com/andrewrk/waveform, 2014.

[39] Khalili, R., Gast, N., Popovic, M., and Le Boudec, J.-Y. MPTCP is not pareto-optimal: Performance issues and a possible solution. *Networking, IEEE/ACM Transactions on 21*, 5 (Oct 2013), 1651–1665.

[40] Khalili, R., Gast, N., Popovic, M., Upadhyay, U., and Le Boudec, J.-Y. MPTCP is not pareto-optimal: Performance issues and a possible solution. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2012), CoNEXT '12, ACM, pp. 1–12.

[41] Klassen, F., and AppNeta. Tcpreplay, 2015. Available at http://tcpreplay.appneta.com/wiki/installation.html#downloads.

[42] Kuhn, N., Lochin, E., Mifdaoui, A., Sarwar, G., Mehani, O., and Boreli, R. Daps: Intelligent delay-aware packet scheduling for multipath transport. In *Communications (ICC), 2014 IEEE International Conference on* (June 2014), pp. 1222–1227.

[43] Lee, K., Lee, J., Yi, Y., Rhee, I., and Chong, S. Mobile data offloading: how much can wifi deliver? In *Proceedings of the 6th International COnference* (2010), ACM, p. 26.

[44] Leech, M., Ganis, M., Lee, Y., Kuris, R., Koblas, D., and Jones, L. Rfc 1928: Socks protocol version 5. Tech. rep., RFC, IETF, March, 1996.

[45] Lim, Y.-s., Chen, Y.-C., Nahum, E. M., Towsley, D., and Gibbens, R. J. How green is Multipath TCP for mobile devices? In *Proceedings of the 4th Workshop on All Things Cellular: Operations, Applications, & Challenges* (2014), ACM, pp. 3–8.

[46] Livadariu, I., Ferlin, S., Alay, O., Dreibholz, T., Dhamdhere, A., and Elmokashfi, A. M. Leveraging the IPv4/IPv6 Identity Duality by using Multi-Path Transport. In *Proceedings of the 18th IEEE Global Internet Symposium (GI)* (Hong Kong/People's Republic of China, Apr. 2015).

[47] Mathis, M., Mahdavi, J., Floyd, S., Romanow, A., and Options, T. S. A. Rfc 2018: Tcp selective acknowledgment options. *Internet Engineering Task Force (IETF)* (1996).

[48] Moon, Y., Kim, D., Go, Y., Kim, Y., Yi, Y., Chong, S., and Park, K. Practicalizing delay-tolerant mobile apps with cedos. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services* (2015), ACM, pp. 419–433.

[49] Nirjon, S., Nicoara, A., Hsu, C.-H., Singh, J. P., and Stankovic, J. A. Multinets: A system for real-time switching between multiple network interfaces on mobile devices. *ACM Transactions on Embedded Computing Systems (TECS) 13*, 4s (2014), 121.

[50] Ostermann, S. Tcptrace. http://tcptrace.org, 2005.

[51] Paasch, C. *Improving Multipath TCP*. PhD thesis, UCL, November 2014.

[52] Paasch, C., Barre, S., et al. Multipath TCP in the Linux Kernel. available from http://www.multipath-tcp.org.

[53] Paasch, C., and Bonaventure, O. Multipath TCP. *Commun. ACM 57*, 4 (Apr. 2014), 51–57.

[54] Paasch, C., Detal, G., Duchene, F., Raiciu, C., and Bonaventure, O. Exploring Mobile/WiFi Handover with Multipath TCP. In *ACM SIGCOMM CellNet workshop* (2012), pp. 31–36.

[55] Paasch, C., Ferlin, S., Alay, O., and Bonaventure, O. Experimental evaluation of Multipath TCP schedulers. In *Proceedings of the 2014 ACM SIGCOMM*

Workshop on Capacity Sharing Workshop (New York, NY, USA, 2014), CSWS '14, ACM, pp. 27–32.

[56] Paasch, C., Khalili, R., and Bonaventure, O. On the benefits of applying experimental design to improve Multipath TCP. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2013), CoNEXT '13, ACM, pp. 393–398.

[57] Peng, Q., Chen, M., Walid, A., and Low, S. Energy efficient Multipath TCP for mobile devices. In *Proceedings of the 15th ACM International Symposium on Mobile Ad Hoc Networking and Computing* (New York, NY, USA, 2014), MobiHoc '14, ACM, pp. 257–266.

[58] Peng, Q., Walid, A., Hwang, J., and Low, S. Multipath TCP: Analysis, design, and implementation. *Networking, IEEE/ACM Transactions on PP*, 99 (2015), 1–1.

[59] Pluntke, C., Eggert, L., and Kiukkonen, N. Saving mobile device energy with Multipath TCP. In *Proceedings of the Sixth International Workshop on MobiArch* (New York, NY, USA, 2011), MobiArch '11, ACM, pp. 1–6.

[60] Postel, J. Rfc793: Transmission control protocol, sept. 1981.

[61] Raiciu, C., Barre, S., Pluntke, C., Greenhalgh, A., Wischik, D., and Handley, M. Improving Datacenter Performance and Robustness with Multipath TCP. In *ACM SIGCOMM 2011* (2011).

[62] Raiciu, C., Niculescu, D., Bagnulo, M., and Handley, M. J. Opportunistic mobility with Multipath TCP. In *Proceedings of the Sixth International Workshop on MobiArch* (New York, NY, USA, 2011), MobiArch '11, ACM, pp. 7–12.

[63] Raiciu, C., Paasch, C., Barre, S., Ford, A., Honda, M., Duchene, F., Bonaventure, O., and Handley, M. 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* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 29–29.

[64] Rao, A., et al. Network characteristics of video streaming traffic. In *CoNEXT '11* (2011).

[65] Rombouts, P. A. pdnsd. http://members.home.nl/p.a.rombouts/pdnsd, 2012.

[66] Semke, J., Mahdavi, J., and Mathis, M. Automatic TCP buffer tuning. *ACM SIGCOMM Computer Communication Review 28*, 4 (1998), 315–323.

[67] Stevens, W. R. Rfc 2001: Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms.

[68] sup Lim, Y., Chen, Y.-C., Nahum, E., Towsley, D., and Lee, K.-W. Cross-layer path management in multi-path transport protocol for mobile devices. In *INFOCOM, 2014 Proceedings IEEE* (April 2014), pp. 1815–1823.

[69] Tachibana, A., Yoshida, Y., Shibuya, M., and Hasegawa, T. Implementation of a proxy-based cmt-sctp scheme for android smartphones. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2014 IEEE 10th International Conference on* (2014), IEEE, pp. 660–665.

[70] Tcpdump/Libpcap. tcpdump, 2015. See http://www.tcpdump.org/.

[71] Team, G. Geotiler. https://github.com/wrobell/geotiler, 2015.

[72] Weaver, N., Kreibich, C., Dam, M., and Paxson, V. Here be web proxies. In *Proceedings of the 15th International Conference on Passive and Active Measurement - Volume 8362* (New York, NY, USA, 2014), PAM 2014, Springer-Verlag New York, Inc., pp. 183–192.

[73] Williams, N., Abeysekera, P., Dyer, N., Vu, H., and Armitage, G. Multipath TCP in Vehicular to Infrastructure Communications. Tech. Rep. 140828A, CAIA, Swinburne University of Technology, August 2014.

[74] Wing, D., and Yourtchenko, A. Happy eyeballs: Success with dual-stack hosts.

[75] Wireshark. Tshark, 2015. See https://www.wireshark.org/docs/man-pages/tshark.html.

[76] Wischik, D., Raiciu, C., Greenhalgh, A., and Handley, M. Design, implementation and evaluation of congestion control for Multipath TCP. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 99–112.

[77] Xiao, Q., Xu, K., Wang, D., Li, L., and Zhong, Y. Tcp performance over mobile networks in high-speed mobility scenarios. In *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on* (2014), IEEE, pp. 281–286.

[78] Xiph.Org. Icecast 2. http://icecast.org, 2004-2015.