Technische Universität München
Lehrstuhl für Kommunikationsnetze
Prof. Dr.-Ing. Wolfgang Kellerer

Master's Thesis

Backward Compatible Multi-Path Routing

Author:	Miñano Belvis, Víctor
Matriculation Number:	03714016
Supervisors:	Babarczi, Péter
	Casademont, Jordi
Begin:	30. October 2018
End:	30. April 2019

With my signature below, I assert that the work in this thesis has been composed by myself independently and no source materials or aids other than those mentioned in the thesis have been used.

München, 30.04.2019
-------------------------------   -------------------------------
Place, Date   Signature

München, 30.04.2019
-------------------------------   -------------------------------
Place, Date   Signature

Abstract

This project studies the behaviour of multipath routing compared to single path routing in order to demonstrate the different benefits that multipath offers.

For this purpose, it have been implemented routers that have 2 routing tables with the capability of storing in one these routing tables the primary next hop for a destination through the shortest path which is calculated by the Open Shortest Path First (OSPF) algorithm, as well as storing a secondary next hop calculated by the Ideal Multipath Routing Expedient (IMRE) algorithm in order to have different paths for the same destination. Besides matching on the destination address, the routers select between the primary and secondary tables based on the Time To Live (TTL) field of the IP header. The end-system can change the forwarding path immediately upon it senses the degradation of the current path by sending the packets with a different TTL value, without waiting for the slow convergence of OSPF to the changed topology.

This multipath behaviour is measured for 3 different use cases. First use case measures the throughput and transmission time when transmitting a file in an ideal scenario where there are no other transmissions at the same time. Second use case performs the measurements for the same transmission as before but when there is a transmission that makes 2 links of the shortest path to be overloaded in order to check the load balancing capability of multipath routing. Finally, the third use case studies the behaviour of multipath routing when there is a failure in a link during the transmission and checks its failure resilience characteristic.

Furthermore, I have studied the paths provided by the IMRE algorithm with a specific TTL match rule. I have demonstrated that in this architecture some TTLs might result in loops, hence, the set of available TTLs for the end-system has to be selected with care.

Contents

Chapter 1 . Introduction......6

Chapter 2 . Background......8

2.1. State-of-the-art......8
2.2. Protocols used......10
2.3. Software used......12

Chapter 3 . Implementation......19

3.1. Routers' behaviour......19
3.2. Creating the Docker images......20
3.3. First topology......23
3.4. TCP scripts......31
3.5. Abilene topology......39

Chapter 4 . Results......42

4.1. Use cases results......42
4.2. Issues with the routing algorithm......51

Chapter 5 . Conclusions......56

5.1. Conclusions about the measurements......56
5.2. Conclusions about the loops in the paths......57
5.3. Future work......57

Appendix A. Multi-socket-TCP sender code......59

Appendix B. Multi-socket-TCP receiver code......65

Appendix C. Paths from "ubuntu-1" to "ubuntu-2" in Abilene topology......71

Appendix D. Paths from "ubuntu-2" to "ubuntu-1" in Abilene topology......74

List of Figures......78

List of Tables......80

Abbreviations **81**

Bibliography **82**

Chapter 1.

Introduction

The traffic on the Internet is increasing more and more every year and thus, applications demand for more and more improvements in the way the data is exchanged between end-users. Multipath can improve this by using the different available paths in the network in the most efficient way. Many networked applications can take benefit of having access to multiple paths between end-users.

Multipath allows the transmission between two end-users to be carried out through different paths at the same time, and thus, making use of the whole capacity of the network. Multipath routing offers many advantages such as load balancing, failure resilience and better transmission throughput since it can be used the whole capacity of the network. However, in practice these carrier networks are not widely deployed because they could need architectural changes such as the application of shim headers, the deployment of middleboxes, and hence, the lack of backwards compatibility.

Nowadays, routing protocols are deployed to perform single path routing. In this type of routing, the routers store in their routing tables the next hop to reach a destination. However, the connection between two end-users may present different paths. By using these single path routing protocols, the whole transmission between two end-users will be carried out through the same path and ignore the others.

The aim of this project is to design an architecture with the ability to perform multipath routing. This architecture has to remain compatible with single path and thus, keeping the backwards compatibility. For this purpose, it is going to be developed and implemented a multipath routing architecture by extending the Open Shortest Path First (OSPF) protocol with the ability to choose between two next hops based on the value of the Time To Live (TTL) value of the IP header.

The idea is to design a router with the ability of having two routing tables with different next hops to the same destination, and implement the corresponding forwarding rules in order to forward an entering packet through the corresponding path depending on the TTL value of the IP header.

Furthermore, it is also necessary to programme a script that has to be run in the end-users and has to perform this multipath behaviour in order to check all the benefits previously said. As the end-users are the ones that choose the TTL value, by implementing this architecture, it is also given more freedom to the end-user.

In Chapter 2 it is explained the background of the project. It is explained the state of the art of this topic, as well as the software used for the implementation.

In Chapter 3 all the implementation process is described in detail and the steps followed to reach the final architecture. Furthermore, it is also explained how the multipath script has been programmed and how it works. This script has also to perform the measurements of the network.

In Chapter 4 all the results of the measurements are shown and commented in order to demonstrate the benefits of multipath.

Finally, in Chapter 5 it is given the conclusions of the project and the future work.

Chapter 2.

Background

In this chapter is explained the state-of-the-art of the multipath topic and different important investigations that have been done so far. Furthermore, it is also explained the way the multipath architecture of this project is going to be carried out, as well as the most important protocols and the software used.

2.1. State-of-the-art

Although multipath routing offers many benefits such as load balancing, failure resilience and better throughput, it is not widely deployed since it can present scalability problems and the lack of backwards compatibility.

However, there are many researches that have study the multipath behaviour for different applications and in different ways [1][2][3][4][7]. Among these works, I am going to highlight 2 of them, which I considered an important baseline for this project.

2.1.1 Deflection routing [1]

In this research the authors' approach is to build an architecture where end-devices set tags to select a path different from the shortest path. Their goal is to find a design that provides the benefit of the source routing but that addresses their associated problems.

In their approach it is not necessary for the end-user to specify which route to take. The end-user just has to be provided by a small number of possible paths and then it can select one among them. The end-users can then test different paths without knowing the routes to which they correspond.

Their approach is based on deflection routing in which routers forward packets off the shortest path when this path is not available. Routers generate these routes by using tags as hints to deflect the packets to neighbours that are not in the shortest path.

The result of their work is to provide end-systems with a high level of path diversity that allows them to avoid undesirable locations within the networks. This scheme is scalable and compatible with Internet Service Providers (ISP) policies because it derives from the deployed internet routing.

They make two contributions. The first one is the use of end-system tags to select path diversity. The second one is the design of routing deflections.

2.1.2 Dynamic Route Computation Considered Harmful [2]

This work sets out a different approach to reduce routing convergence. The authors assume that changes in the topology are less frequent than changes in the status of a link. They aim to separate the routing computation from the failure handling.

They propose computing paths on the deployed topology. As they assume that the changes in the topology are much less frequent than changes in the link state, there can be calculated a great variety of routes and thus, they assume that the number of alternative routes calculated is enough to mask failures. When there is a failure in a link (the link state changes to down), the packets can be sent through another path that is not affected by the failure.

Their approach relies on conventional mechanisms to perform packet forwarding, but decouples the route selection process from routers. This means that the route selection process is performed in the end-hosts by just changing some bits of the header. End-devices have to implement application-level failure recovery. They also have to detect endpoint-based failure. To select the path they monitor the availability and quality of paths and use this to inform their choice.

2.1.3 Conclusions

I wanted to detail these papers since I think both have some important clues for my final architecture. They both propose a system where it is given freedom to the end-user since it is able to decide which path select among different paths. This path selection is made by simply changing some bits of the header or by using tags.

In my approach I am going to use this principle to design a scalable architecture. I am going to let the end-user to select the path(s) to transmit the data by just changing the TTL value of the IP header.

Another important clue extracted from the papers is that they both propose calculating the different paths from the already-calculated shortest path.

In my design, I am going to use a software that provides the Open Shortest Path First (OSPF) algorithm to calculate the shortest path. This shortest path will be the primary path for every router. The secondary path will be calculated based on this shortest path by using another algorithm called Ideal Multipath Routing Expedient (IMRE) algorithm.

With these two mechanisms: the selection of the path depending on the TTL value and the way of calculating the different paths, my architecture will be scalable and backwards compatible.

2.2. Protocols used

In the introduction of the project, as well as in the state-of-the-art part, I have mention two important protocols that I am going to use throughout this thesis and I consider important to explain.

2.2.1 Transmission Control Protocol (TCP)

The most used protocol on the Internet is TCP. TCP is a connection oriented protocol. It uses flow and error control mechanisms at transport layer and thus, it is a reliable transport protocol.

TCP ensures that the data will reach the destination without errors. Additionally, it also ensures that the data sent is received in the same order that it was transmitted. TCP uses IP addresses and port numbers to identify to which connection the payload belongs to. TCP also permits monitoring the data flow in order to avoid the saturation of the network.

2.2.1.1 Control plane

TCP splits the byte stream sent in smaller segments and transmits them over the underlying IP protocol. The IP header has the source and destination IP addresses. In order to identify to which connection the payload belongs to, apart from the IP addresses, it is also necessary to use source and destination port numbers.

In order to establish a TCP connection, the hosts go through the 3-way handshake which is a three-step method that requires both hosts to exchange SYN and ACK packets before the data is started to be sent.

The 3-way handshake works in the following way:

- The client sends a SYN data packet to the server
- The server must have open ports that can accept and initiate new connections. When it receives the SYN packet, it answers by sending a SYN/ACK packet
- Finally, the client responds with an ACK packet

After this procedure, the connection is established, and the data can be transmitted.

When the data transmission is completed and the client wants to close the connection, it sends a segment with the FIN flag enabled and enters in a FIN-WAIT state. While the client device is in FIN-WAIT state, it can continue receiving and processing data from the server device, but it does not send more data. When the server also finishes its transmission, it also sends the segment with the FIN flag enabled. The client then replies with an ACK and both devices finish the connection.

Figure 2.1 shows all this TCP procedure from establishing the connection, the data transmission, and the end of the connection.

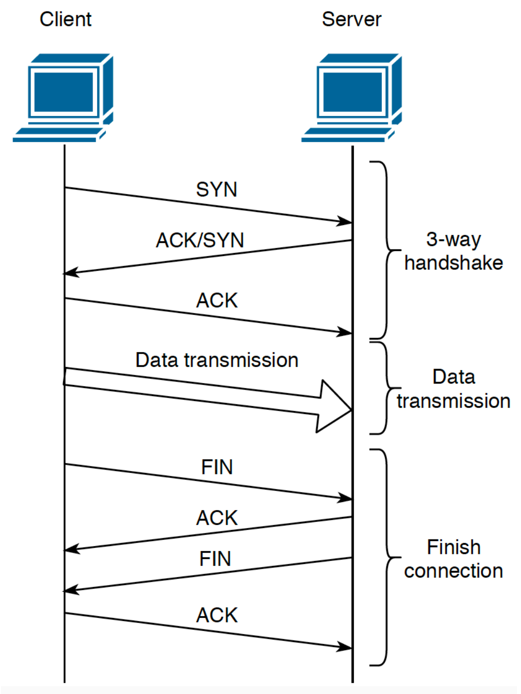


Figure 2.1. TCP connection

2.2.1.2 Data plane

TCP assigns a sequence number to each byte of the data stream in order to ensure the reliable and in-order delivery of the byte stream. The sequence number is included inside the TCP header. To allow the reliable transmission, the receiver sends acknowledgments back to the sender. These acknowledgements specify the following sequence number that the receiver is expecting to receive.

TCP also has a congestion window which dynamically adapts the sending rate of a TCP connection in order to use the whole capacity of the network. In each TCP segment, the receiver specifies the amount of bytes that can store in its buffer for that connection.

2.2.2 Open Shortest Path First (OSPF) Protocol

OSPF a link-state routing protocol described in RFC 2328. It uses the Dijkstra algorithm to calculate the shortest path between 2 nodes. It uses the "cost" as the metric value which takes into account different parameters such as the bandwidth and link congestion. OSPF builds a link state data base (LSDB) identical for all the routers in the same area.

Some of the most important characteristics of OSPF are the following:

- It is a classless protocol. This means that supports Variable Length Subnet Mask (VLSM) and Classless Inter-Domain Routing (CIDR).
- It has a fast convergence since it quickly propagates network changes.
- It is scalable. It works well in small, as well as in large network sizes. It supports a hierarchical system since routers can be grouped into areas
- It is secure. It supports Message Digest 5 (MD5) authentication. When this feature is enabled, the routers only accept encrypted routing updates from the routers with the same pre-shared password

2.2.3 Conclusions

TCP protocol has been chosen for the architecture because it is needed a reliable protocol for the transmission in order to preserve the integrity of the file transmitted.

OSPF protocol has to be used because the IMRE algorithm reads the OSPF LSDB in order to have a global view of the topology and calculate the secondary paths. The IMRE algorithm is explained in Section 2.3.5.

2.3. Software used

In this section it is explained the software utilised to carry out the project.

2.3.1 GNS3

The software employed to simulate the topologies is GNS3 (Graphical Network Simulation 3). GNS3 is an open source, free network software emulator which is used to emulate, configure, test and troubleshoot virtual and real networks.

Last versions of GNS3 support devices from multiple network vendors such as Cisco virtual switches, Cisco ASAs, Brocade vRouters, Cumulus Linux switches, Docker instances, HPE VSRs, multiple Linux appliances and many others [9]. Furthermore, GNS3 offers no limitation on the number of devices supported. The only possible limitations are in the CPU and memory of the hardware that runs it.

GNS3 emulates the hardware of a device and runs real images in the virtual device, so it can be used to design complex networks and do simulations about them. Since it runs real images, it is necessary to have the images of the devices to be simulated.

GNS3 offers on his official website different appliances that have and already configured image and can be used to emulate a device.

For the realization of this project it is needed an image for the routers and another image for the hosts.

2.3.1.1 GNS3 Architecture

GNS3 consists of two software components [10]:

- Client part: The GNS3-all-in-one software (GUI)
- Server part: The GNS3 virtual machine (VM)

The GNS3-all-in-one is the graphical user interface (GUI) where the topologies can be created.
When the topologies are created, the devices are hosted and run by a server process. The options for the server part are the following:

- **Local GNS3 server:** run on the same PC where the GUI is installed.
- **Local GNS3 VM:** run on the same PC using virtualization software such as VMware or Virtualbox.
- **Remote GNS3 VM:** run remotely using VMware ESXi or in the cloud.

For this project I chose to host the devices on the GNS3 VM using VMware which is the recommended option.

2.3.2 Docker

Docker is an open platform that can be used for developing, shipping and running applications by using containers. Containers offer so many benefits since they are flexible, lightweight, interchangeable, portable, scalable and stackable. Containers are run directly in the kernel of the host by running an image [11].

An image is a read-only template with instructions for creating a container. To build an image it is necessary to create a file called Dockerfile which contains a simple syntax for defining the steps to create the image with the necessary packages and run it.

A container is a runnable instance of an image. By using the Docker API or the CLI, a container can be created, started, stopped, moved, or deleted. When a container is deleted, all the changes that have been performed in it disappear. Therefore, when you want to make the changes permanent, you have to save the container into an image which can be run again in another container.

In this project, the images needed to be run using the containers are two: one for the hosts and other for the routers.

2.3.3 iptables

“iptables” is a command line utility used to configure the Linux kernel firewall. “iptables” can be used to inspect, modify, forward, redirect, or drop IP packets. It can be used to modify and mark the packets and thus, affecting packet forwarding.

“iptables” contains five built-in tables: raw, filter nat, mangle, security. Each table contains a number of chains. The chains contain rules that specify the corresponding action that has to be performed with a packet that matches.

With “iptables” command, the user can modify these chains and rules. Therefore, this is an important feature to modify the routers behavior.

When a packet is received on any of the interfaces, it goes through the chains of the tables. The order that the packet follow is show in Figure 2.2. [12].

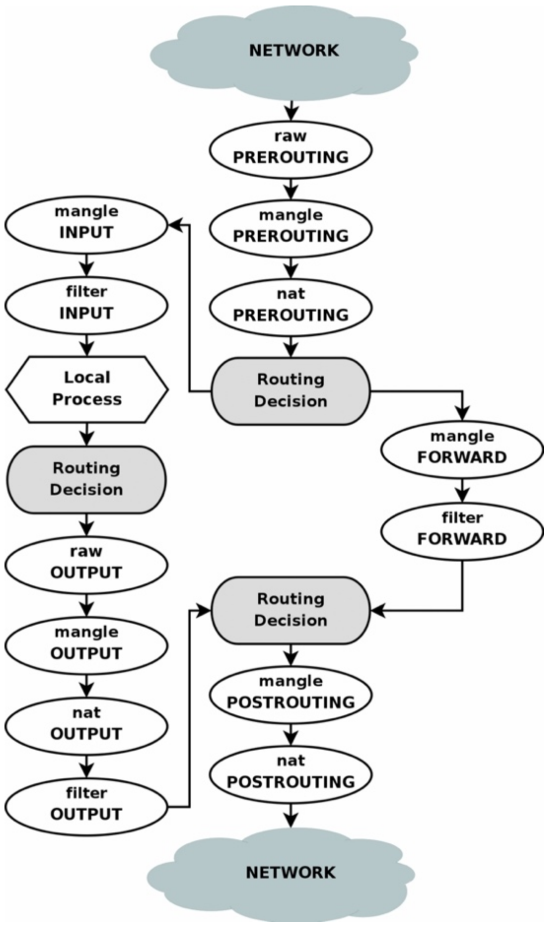


Figure 2.2. iptables routing and packet filtering process [12]

2.3.3.1 Chains

There are five built-in chains [13]:

- **PREROUTING:** for altering packets in the moment they arrive before making any routing decisions.
- **INPUT:** for altering packets that are destined to local sockets.
- **OUTPUT:** for altering packets that are generated locally.
- **FORWARD:** for altering packets that are routed through the box.
- **POSTROUTING:** for altering packets that go into the network after all routing decisions have been made.

Despite these are built-in chains, the user can also create their own chains.

2.3.3.2 Tables

The built-in tables that iptables contain are listed below. The built-in chains that each table contain are also given [13].

- **raw table:** raw table filters the packets before any other table. It is used mainly to configure the exemptions from connection tracking. The built-in chains it contains are: PREROUTING and OUTPUT.

- **filter table:** it is the default table. The built-in chains it contains are: INPUT, FORWARD and OUTPUT.

- **nat table:** it is used for network address translation. The built-in chains it contains are: PREROUTING, INPUT, OUTPUT and POSTROUTING.

- **mangle table:** it is used for special alterations of packets. The built-in chains it contains are: PREROUTING, OUTPUT, INPUT, FORWARD and POSTROUTING.

- **security table:** it is used for Mandatory Access Control (MAC) networking rules. This table is called after the filter table. The built-in chains it contains are: INPUT, OUTPUT and FORWARD.

2.3.4 FRR (Free Range Routing)

FRR is a routing software package for Linux and Unix platforms. It is similar to Quagga, but contains several extensions to provide the best routing protocol stack available. It provides TCP/IP based routing services with support for routing protocols such as BGP, IS-IS, LDP, OSPF, PIM, and RIP.

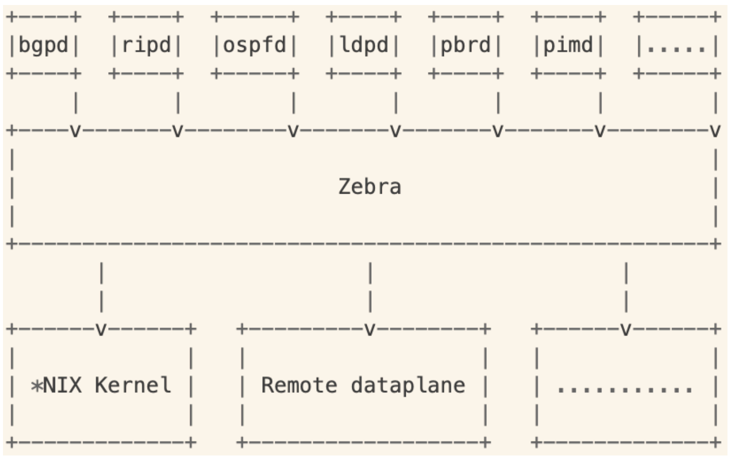
A device with FRR installed acts as a dedicated router. It offers an interactive user interface for each routing protocol and supports common client commands. There are two user modes: the normal mode and the enable mode. In normal mode the user can only view system status, whereas in enable mode the user can change the system configuration.

2.3.4.1 FRR architecture

FRR is a suite of daemons that work together to build the routing table. There is a daemon for each routing protocol. Besides there is another daemon called Zebra which acts as an intermediary between the other daemons and the kernel. Figure 2.3 shows the FRR architecture.

Since each daemon runs independently, if there is a failure in a daemon, it does not affect the others. Therefore, the FRR architecture allows high resiliency and flexibility. This modularity makes it easy to implement new protocols, so it is also extensible.

All these daemons can be managed through an user interface called "vtysh" which also provides the ability to configure all the daemons by using a single configuration file.



```
+----+  +----+  +-----+  +----+  +----+  +----+  +-----+
|bgpd|  |ripd|  |ospfd|  |ldpd|  |pbrd|  |pimd|  |.....|
+----+  +----+  +-----+  +----+  +----+  +----+  +-----+
     |       |        |       |       |       |        |
+----v-------v--------v-------v-------v-------v--------v
|                                                      |
|                        Zebra                         |
|                                                      |
+------------------------------------------------------+
       |                   |                  |
       |                   |                  |
+------v------+   +---------v--------+   +------v------+
|             |   |                  |   |             |
| *NIX Kernel |   | Remote dataplane |   | ........... |
|             |   |                  |   |             |
+-------------+   +------------------+   +-------------+
```

Figure 2.3. FRR architecture [14]

2.3.4.2 OSPF daemon

As previously said, OSPF is a link-state routing protocol described in RFC 2328. ospfd (OSPF daemon) must get the information from zebra, therefore, zebra must be running before invoking ospfd. The ospfd configuration is done in the configuration file "ospfd.conf".

2.3.4.3 OSPF API

OSPF API is contained in the OSPF daemon. It provides retrieval of the link-state database (LSDB) of the OSPF daemon by allowing external application to obtain a copy of this database which includes router and network link-state advertisements (LSAs). When a new LSA arrives at the OSPF daemon, the application is informed by the API module by sending a message and thus, the application is always synchronized with the LSDB.

OSPF API also allows the origination of own opaque LSAs which are distributed in a transparent way to the other routers.

OSPF API architecture

Figure 2.4 shows the OSPF API architecture. The OSPF core module executes the OSPF protocol which discovers neighbours and exchange neighbour state. The opaque module allows the exchange of opaque LSAs between routers. These opaque LSAs can be generated by modules like MPLS-TE module or the API server module. The API server module listens to connections from external applications that want to communicate with the OSPF daemon and can handle multiple clients at the same time.

The client external application links against the OSPF API client library which establishes a socket connection with the API server module and uses this connection to get the LSAs and originate opaque LSAs.

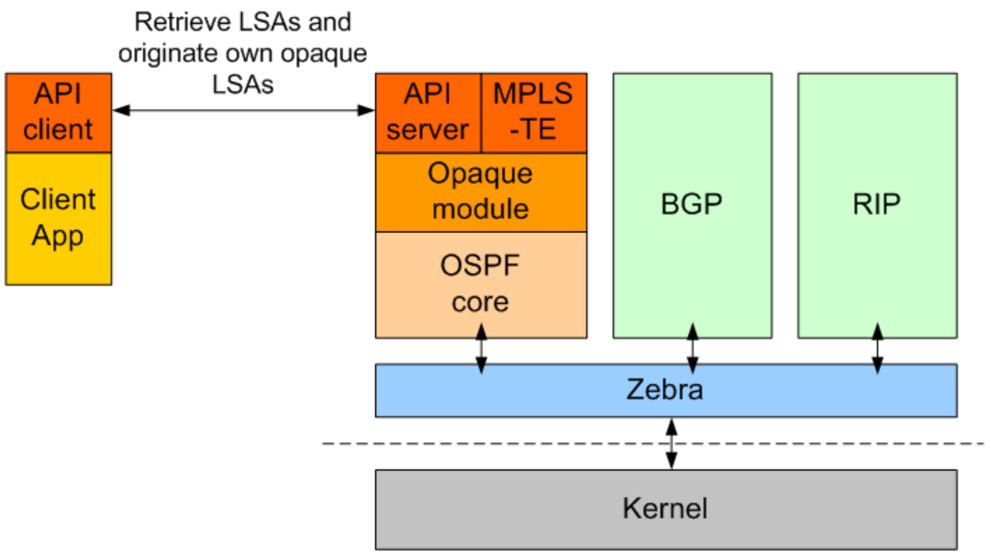


Figure 2.4. OSPF API architecture [15]

2.3.5 IMRE algorithm

All the information about the IMRE algorithm is extracted from [5]. IMRE is a labelling-based routing scheme. One of its most important features is that is compatible with existing link-state routing protocols such as OSPF. It offers alternative paths as short as possible to a destination host. It uses the TTL field of the IP header to label the packets that are going to be sent through one path or the another.

Having multiple paths to reach a destination has benefits such as more reliable connection, better utilization of the network and providing connections with different properties. On the other hand, it presents the disadvantage of requiring more routing table entries and computing power.

IMRE retrieves the LSDB information of the OSPF daemon of FRR through the OSPF API client. With this information it has a global view of the topology and the shortest paths calculated by the OSPF daemon which are stored in the routers. Making use of this information it executes the algorithm in order to calculate the following shortest path. Then the OSPF shortest path is stored in a primary routing table of the routers, and the following shortest path calculated by the IMRE algorithm is stored in a secondary routing table of the routers.

Chapter 3.

Implementation

In this chapter all the steps to configure the software and build the topology are explained.

3.1. Routers' behaviour

The most important thing for the topology to work in the desired way, is the routers' behaviour. Routers are in charge of storing the routing tables and taking the forwarding decisions.

Figure 3.1 shows the global view of how the routers should work. This figure is based on a figure of [5], but it has been slightly modified in order to adapt it to this project.

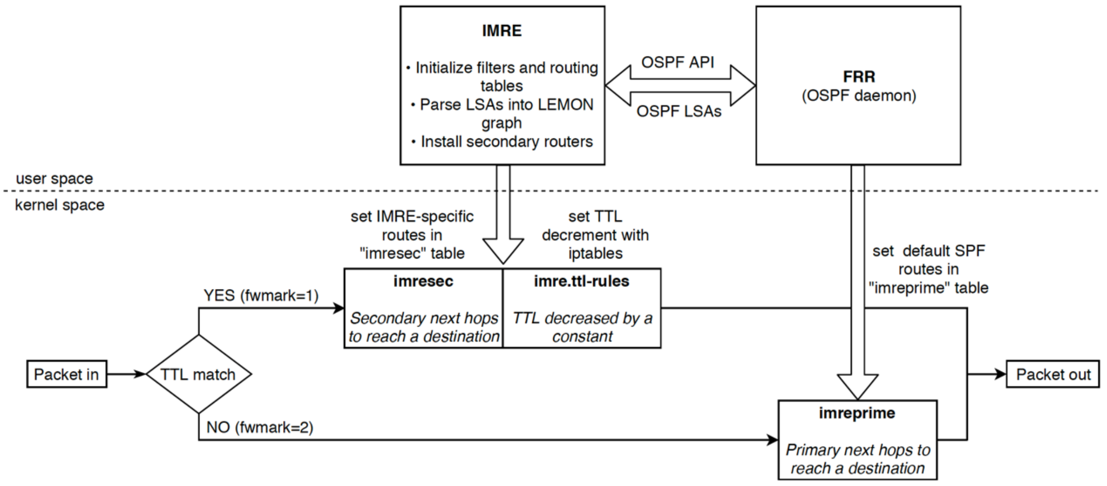

IMRE
• Initialize filters and routing tables
• Parse LSAs into LEMON graph
• Install secondary routers
OSPF API
OSPF LSAs
FRR
(OSPF daemon)
user space
kernel space
set IMRE-specific routes in "imresec" table
set TTL decrement with iptables
set default SPF routes in "imreprime" table
imresec
Secondary next hops to reach a destination
imre.ttl-rules
TTL decreased by a constant
YES (fwmark=1)
Packet in
TTL match
NO (fwmark=2)
imreprime
Primary next hops to reach a destination
Packet out

Figure 3.1. Global view of router behaviour [5]

First of all, 2 additional routing tables are added to each router. These routing tables are "imreprime" and imresec". Afterwards, the rules for marking the packets are created with the command "iptables". Then the forwarding rules depending on the mark of the packet are created with the command "ip rule".

The FRR block executes the OSPF algorithm to calculate the shortest path. It also has a global view of the protocol area. As all the routers in the topology that I am going to create are going to be in the same area, each router is going to have a global view of the topology.

The IMRE block, reads the OSPF daemon LSDB through the OSPF API that FRR offers. It stores the shortest path calculated by the OSPF daemon into the “imreprime” routing table of the router. Furthermore, it calculates the following shortest path to the same destination, and stores this secondary next hop it in the “imresec” routing table of the router.

With all this configuration, when a packet enters in the router, it examines its TTL value and marks the packet with the corresponding value. Then the router checks the next hop in “imresec” or “imreprime” depending on the mark of the packet. Finally, it forwards the packet to the corresponding next hop.

As said in Chapter 2, this router configuration, as well as the whole design of the topology, is going to be performed with the GNS3 software. By using this software, I am going to build the topology, emulate the devices and obtain the results is GNS3.

Therefore, the first step is to install both GNS3 and the GNS3 VM. They both can be downloaded from the GNS3 official webpage.

The way I implemented all this procedure is explained in the next sections.

3.2. Creating the Docker images

Before being able to build the topologies, it is necessary to create the device images. GNS3 offers in its webpage different appliances of devices that can be used. Unfortunately, I found it challenging to install in them the rest of the software needed. Due to these limitations I decided not to use these appliances, but to make my own images.

A good solution that I found is to create Docker images and run them in the Docker containers.

The first step to create Docker images is to create a file called “Dockerfile” which contains all commands needed to build the image. Docker builds the image by reading the instructions from this file.

When creating a Docker image, all the packages to be installed can be defined in the “Dockerfile”. Packages can be also installed after creating the image, but there must be created a copy of this new image if you want the new packages to remain and do not be lost when stopping the container.

3.2.1 Docker image for hosts

In order to build the Docker image for the hosts, I added the basic needed packages in the "Dockerfile". These packages include the commands for testing or changing the routing rules.

The content of the "Dockerfile" I used to create the images is the following:

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install -y nmap \
        sudo psmisc nano wget lsb \
        net-tools iputils-ping iproute2 iptables udhcpc ethtool
```

As it can be seen, the image will have the Operating System (OS) Ubuntu 14.04. I chose this OS because it is easy to install FRR on it and all the necessary packages can be installed on it.

The networking packages installed are the following:

- **net-tools:** needed to assign the IP addresses to the interfaces, as well as adding the routes,
- **iputils-ping:** needed to test the connection from one device to other,
- **iproute2:** package needed to add the routes and the rules for marking the packets,
- **iptables:** needed for the forwarding rules,
- **udhcpc:** it is useful to assing automatically an IP address to an interface,
- **ethtool:** useful to check the link speed.

When having created the "Dockerfile", next step is to build the image by running the following command:

```
docker build [OPTIONS] PATH | URL | -
```

And in this case:

```
sudo docker build -t ubuntu .
```

Where "-t" option allows you to add a name for the image. In this case I chose the name "ubuntu" since it is the OS chosen for the image. The point at the end of the command indicates the path where the "Dockerfile" is. In this case I run the command in the location where actually the file was.

Once the image is created, it can be added to GNS3 and will be available for building the topologies.

3.2.2 Docker image for routers

The router image has to include the same packages as the host image but, in addition, it has to include the FRR software and the IMRE algorithm. So, taking as a base image the one previously created, both FRR and IMRE have to be installed and saved a new copy of the new image with FRR installed.

To run the image in a new container it has to be executed the following command:

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

And in this case:

```
docker run -i -t ubuntu:latest /bin/bash
```

Where "-i" option makes the image run in interactive mode, and "-t" option allocates a pseudo-TTY. "ubuntu:latest" is the image created for the host where "latest" is a tag that indicates that it has to run the latest version created for that image name.

3.2.2.1 FRR and IMRE installation

As said in Chapter 2, IMRE algorithm gets the LSDB information of the OSPF daemon of FRR through the OSPF API client. For this purpose, IMRE uses the FRR original files that can be downloaded from the FRR official webpage **Error! Reference source n ot found.**, but with a modified "ospfclient" file used to access the needed information.

Therefore, to install both FRR and IMRE it is necessary the FRR files with the modified "ospfclient" file, as well as, the files that contain the IMRE algorithm.

To copy files or directories from the GNS3 VM to a container or vice versa, it can be used the "docker cp" command:

```
docker cp [OPTIONS] CONTAINER:SRC_PATH DEST_PATH|-
docker cp [OPTIONS] SRC_PATH|- CONTAINER:DEST_PATH
```

Where "CONTAINER" is the identification number of the container. To check the identification number of the containers that are running, it can be used the "docker ps" command.

When the container has the required files, the following step is to install FRR. It can be followed the steps that are indicated on its webpage. Once FRR is installed, the router is prepared to act as a dedicated router with the FRR and IMRE capabilities.

Next step is to store this new configuration into a new Docket image using the "docket commit" command:

```
docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]
```

Where “CONTAINER” is the container id that you want to store in a new image. “REPOSITORY” is the name for this new image. In this case, I chose the name “frr-ubuntu” for this router image.

Now this image can be also added to GNS3 and thus, we have one “ubuntu” image for hosts and one “frr-ubuntu” image for routers.

3.3. First topology

First of all, I decided to create a simple topology with 3 routers and 2 hosts. The objective of building such a simple topology is to check if FRR and IMRE work properly. Another aim is to install the forwarding rules in the routers with “iptables” and check if the packets follow its corresponding path depending on its TTL value.

The tasks for this aim are the following:

- Create the simple topology with 3 routers and 2 hosts.
- Add 2 new routing tables in the routers: primary and secondary.
- Introduce the forwarding rules [8].
- Run FRR and IMRE in the routers.
- Check that the paths stored in the routers are correct.
- Study the behaviour of the topology.

The topology designed can be seen in Figure 3.2.

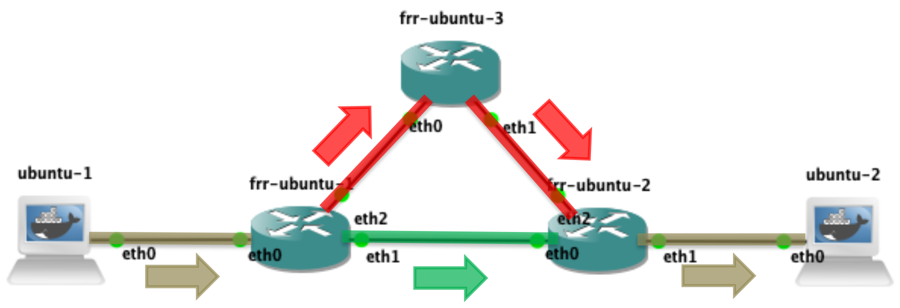


Figure 3.2. Simple topology

Once the topology is created, next step is to add the routing tables. The routing tables have to be added to the file “rt_tables” which is in the path “/etc/iproute2/rt_tables”. As IMRE suggests, the primary routing table is “imreprime” and the secondary routing table is “imresec”:

```
echo 15 imresec >> /etc/iproute2/rt_tables
echo 10 imreprime >> /etc/iproute2/rt_tables
```

Next step is setting the forwarding rules. I used the forwarding rules that IMRE suggests. These rules have been created on the “mangle” table of the “PREROUTING” chain and indicate that when the packet TTL value is “1***000”, where * can be 0 or 1, are marked with value “1”, and marked with value “2” in the rest of cases. Packets marked with value “1” must follow the secondary routing table, whereas packets marked with value “2” must follow the primary routing table. Table 3.1 shows this behaviour. The TTL of the packets that match the matching rule, and are forwarded through the secondary path, is decreased by a constant

Destination	*TTL*	*Mark*	*Action*
r	1 *** 0000	1	Forward to imresec; TTL– Constant
r	* *** ****	2	Forward to imreprime; TTL– –

Table 3.1. Forwarding rules

And hence, the TTL values that match these forwarding rules are the ones shown in Table 3.2. The binary digits represented in red are the ones that change.

Binary value	*Decimal value*
1000 0000	128
1001 0000	144
1010 0000	160
1011 0000	176
1100 0000	192
1101 0000	208
1110 0000	224
1100 0000	240

Table 3.2. Matching TTL values

To add the matching rules to mark the packets I used “iptables”:

iptables -t mangle -A PREROUTING -m u32 --u32 "5&0x8F=0x80" -j MARK --set-mark 1
iptables -t mangle -A PREROUTING -m u32 ! --u32 "5&0x8F=0x80" -j MARK --set-mark 2

Then, I set the forwarding rules depending on this mark with the command “ip rule”:

ip rule add fwmark 1 table imresec prio 1000
ip rule add fwmark 2 table imreprime prio 1001

FRR uses the daemon configuration files to configure the network. Zebra daemon reads from the “zebra.conf” file the IP addresses of each interface. OSPF daemon reads from the “ospfd.conf” file all the OSPF configuration which has to include the network of each router interface and their area. The daemon configuration files are in the directory “/etc/frr”.

For example, in this simple topology the content of “zebra.conf” of router “frr-ubuntu-1” is the following:

```
-*- zebra -*-
hostname R1

!interfaces
interface eth0
  ip address 10.0.0.1/24

interface eth1
  ip address 10.1.0.0/24

interface eth2
  ip address 10.3.0.1/24

!end interfaces

log file /tmp/zebrad.log
```

And the content of “ospfd.conf” of “frr-ubuntu-1” is the following:

```
! -*- ospf -*-
hostname ospfd-R1

!interfaces
interface eth0
  ip ospf network point-to-point

interface eth1
  ip ospf network point-to-point

interface eth2
  ip ospf network point-to-point

!end interfaces

router ospf
        ospf router-id 10.0.0.1

!transit networks
        network 10.0.0.0/24 area 0.0.0.0
        network 10.1.0.0/24 area 0.0.0.0
        network 10.3.0.0/24 area 0.0.0.0
!transit networks end

log file /tmp/ospfd.log
```

Once the configuration files are created and stored in “/etc/frr”, FRR can be run. To run it, the best way is to run each daemon separately. As OSPF daemon needs Zebra daemon to be running, the first daemon to run is the Zebra daemon. To run zebra, the next command has to be executed in each router:

```
/usr/lib/frr/zebra -d -f /etc/frr/zebra.conf
```

Where -d option indicates that it has to be run in daemon mode, and -f option is used to indicate the path where the configuration file is.

When the Zebra daemon is running, it can be run now the routing protocol daemons. In this case it is only necessary the OSPF daemon:

```
/usr/lib/frr/ospfd -a -d -f /etc/frr/ospfd.conf
```

Where -a option enables the OSPF API server.

After executing the previous commands, FRR has to be running and the OSPF daemon calculates the shortest paths in every router and stores it in the main routing table. Next step now is to run the IMRE algorithm to calculate the secondary paths and store both the shortest path and the secondary path in the "imreprime" and "imresec" routing tables respectively.

As said in Chapter 2.3.5, IMRE algorithm gets the LSDB information of the OSPF daemon through the OSPF API. It includes a modified "ospfclient" file that gets this information and runs the algorithm [5]. So, what has to be done to run this algorithm is to run this "ospfclient" file with the following command:

```
/frr/ospfclient/ospfclient --hostname localhost --strun /ST/st-demo --stfile /outputs/multipath.lgf --undofile /outputs/multipath.undo -d -i /run/frr/ospfclient.pid
```

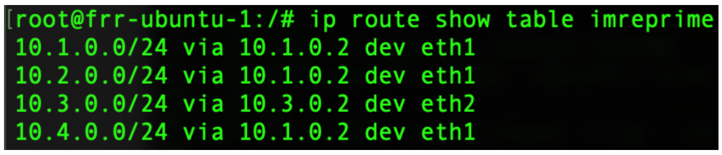
Where "--strun" option indicates the path where the IMRE algorithm is stored.

"--stfile" indicates the path where the current topology has to be stored in LEMON graph format [16]. This file is used by the IMRE algorithm to calculate the secondary paths.

"--undofile" indicates the path where it has to be stored a file with the necessary commands to delete the routes that are added to the routing tables. When running the IMRE algorithm, all the current routes are deleted and new ones are stored. In order to delete the routes added previously, this file is necessary.

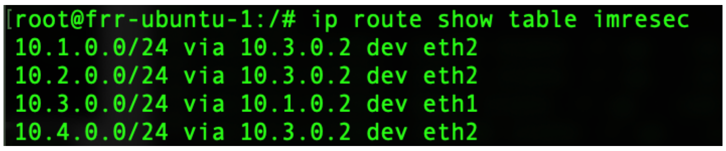
"-i" option indicates the path to store the process identification number (pid).

After all this procedure, all the routers have to be properly configured with the routing protocols and the forwarding rules. Figure 3.3 (a) and (b) show the "imreprime" and "imresec" routing tables of the router "frr-ubuntu-1". It can be seen that the routes have been correctly stored.



```
[root@frr-ubuntu-1:/# ip route show table imreprime
10.1.0.0/24 via 10.1.0.2 dev eth1
10.2.0.0/24 via 10.1.0.2 dev eth1
10.3.0.0/24 via 10.3.0.2 dev eth2
10.4.0.0/24 via 10.1.0.2 dev eth1
```

(a) “imreprime” routing table



```
[root@frr-ubuntu-1:/# ip route show table imresec
10.1.0.0/24 via 10.3.0.2 dev eth2
10.2.0.0/24 via 10.3.0.2 dev eth2
10.3.0.0/24 via 10.1.0.2 dev eth1
10.4.0.0/24 via 10.3.0.2 dev eth2
```

(b) “imresec” routing table

Figure 3.3. “frr-ubuntu-1” routing tables

3.3.1 First topology check

When finished the configuration of all the devices, next step is to check if the network behaves the desired way. The tasks that have been followed to perform this verification are the following:

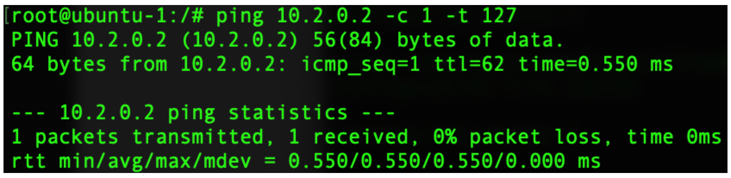
- Send a ping from host “ubuntu-1” to the host “ubuntu-2” with TTL different to “1***000” and check with Wireshark if it follows the primary path.
- Send a ping form host “ubuntu-1” to the host “ubuntu-2” with TTL equal to “1***000” and check with Wireshark if it follows the secondary path. One value equal to “1***000” is for example “128” which in binary is “1000000”.

3.3.1.1 Ping through primary path

First of all, a Wireshark has to be listening in network between routers “frr-ubuntu-1” and “frr-ubuntu-2”. Afterwards, to send the ping from the host “ubuntu-1” to the host “ubuntu-2” I used the command “ping”:

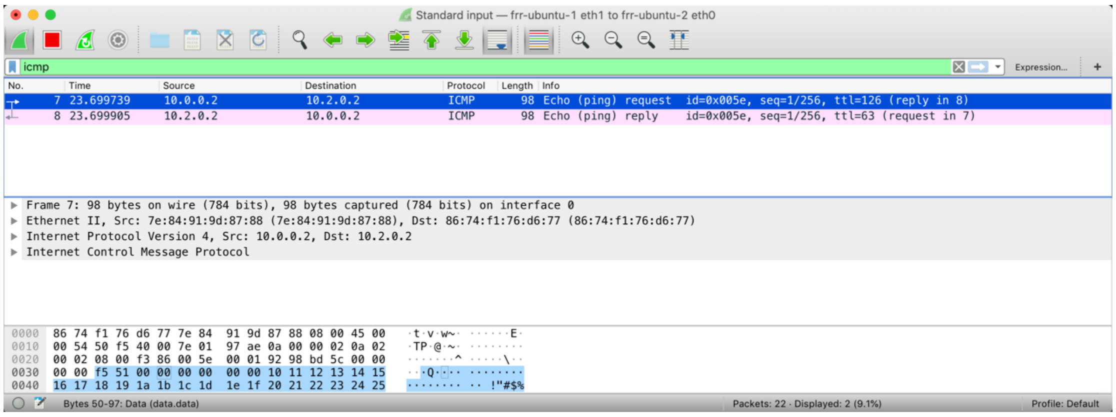
ping 10.2.0.2 -c 1 -t 127

Figure 3.4 shows a screenshot of the console of “ubuntu-1” with the ping outcome. Figure 3.5 shows the ping in Wireshark. It can be seen that the ping has been successful and has follow the correct path.



```
[root@ubuntu-1:/# ping 10.2.0.2 -c 1 -t 127
PING 10.2.0.2 (10.2.0.2) 56(84) bytes of data.
64 bytes from 10.2.0.2: icmp_seq=1 ttl=62 time=0.550 ms

--- 10.2.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.550/0.550/0.550/0.000 ms
```

Figure 3.4. "ubuntu-1" console with the ping through the primary path



```
Standard input — frr-ubuntu-1 eth1 to frr-ubuntu-2 eth0
icmp
No.  Time       Source     Destination  Protocol  Length  Info
  7  23.699739  10.0.0.2   10.2.0.2     ICMP         98   Echo (ping) request  id=0x005e, seq=1/256, ttl=126 (reply in 8)
  8  23.699905  10.2.0.2   10.0.0.2     ICMP         98   Echo (ping) reply    id=0x005e, seq=1/256, ttl=63 (request in 7)

▶ Frame 7: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
▶ Ethernet II, Src: 7e:84:91:9d:87:88 (7e:84:91:9d:87:88), Dst: 86:74:f1:76:d6:77 (86:74:f1:76:d6:77)
▶ Internet Protocol Version 4, Src: 10.0.0.2, Dst: 10.2.0.2
▶ Internet Control Message Protocol

0000  86 74 f1 76 d6 77 7e 84  91 9d 87 88 08 00 45 00   ·t·v·w~· ······E·
0010  00 54 50 f5 40 00 7e 01  97 ae 0a 00 00 02 0a 02   ·TP·@·~· ········
0020  00 02 08 00 f3 86 00 5e  00 01 92 98 bd 5c 00 00   ·······^ ·····\··
0030  00 00 f5 51 00 00 00 00  00 00 10 11 12 13 14 15   ···Q···· ········
0040  16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25   ········ ·· !"#$%
Bytes 50-97: Data (data.data)          Packets: 22 · Displayed: 2 (9.1%)          Profile: Default
```

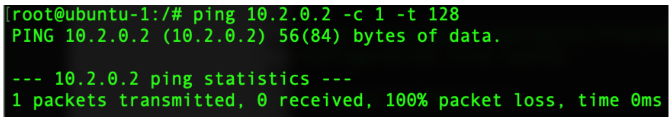
Figure 3.5. Wireshark showing the ping through the primary path between routers "frr-router-1" and "frr-router-2"

3.3.1.2 Ping through secondary path

In this case, Wireshark has to be listening in the link between "frr-router-1" and "frr-router-3" in order to verify that the packet follows the secondary path. Afterwards, to send the ping from the host "ubuntu-1" to the host "ubuntu-2" I used again the "ping" command but establishing a TTL of 128.

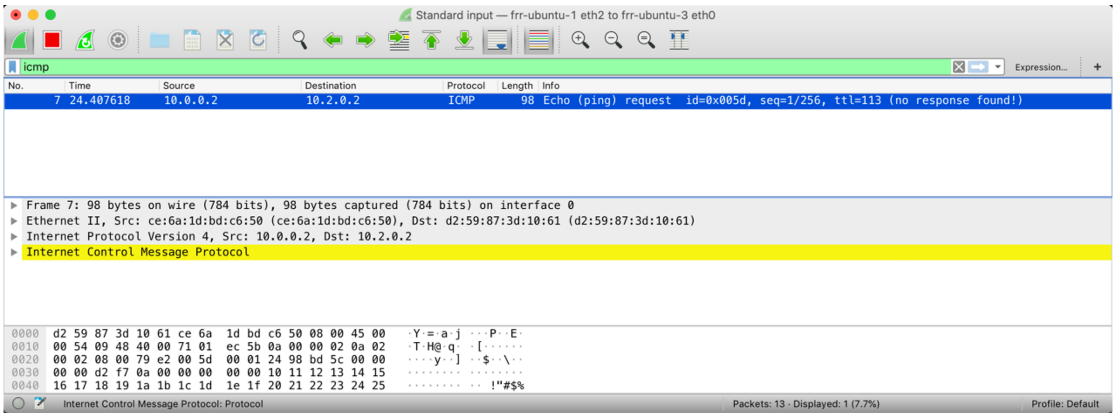
ping 10.2.0.2 -c 1 -t 128

Figure 3.6 shows a screenshot of the console of "ubuntu-1" with the ping outcome where it can be seen that the ping was unsuccessful. Figure 3.7 shows the ping in Wireshark where it can be seen that the packets try to follow the expected path, but it only shows the ping request but not the response.



```
root@ubuntu-1:/# ping 10.2.0.2 -c 1 -t 128
PING 10.2.0.2 (10.2.0.2) 56(84) bytes of data.

--- 10.2.0.2 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

Figure 3.6. "ubuntu-1" console with the unsuccessful ping through the secondary path



Standard input — frr-ubuntu-1 eth2 to frr-ubuntu-3 eth0

icmp

```
No.   Time        Source      Destination  Protocol  Length  Info
    7 24.407618   10.0.0.2    10.2.0.2     ICMP          98  Echo (ping) request  id=0x005d, seq=1/256, ttl=113 (no response found!)
```

```
▶ Frame 7: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
▶ Ethernet II, Src: ce:6a:1d:bd:c6:50 (ce:6a:1d:bd:c6:50), Dst: d2:59:87:3d:10:61 (d2:59:87:3d:10:61)
▶ Internet Protocol Version 4, Src: 10.0.0.2, Dst: 10.2.0.2
▶ Internet Control Message Protocol
```

```
0000  d2 59 87 3d 10 61 ce 6a  1d bd c6 50 08 00 45 00   ·Y·=·a·j ···P··E·
0010  00 54 09 48 40 00 71 01  ec 5b 0a 00 00 02 0a 02   ·T·H@·q· ·[······
0020  00 02 08 00 79 e2 00 5d  00 01 24 98 bd 5c 00 00   ····y··] ··$··\··
0030  00 00 d2 f7 0a 00 00 00  00 00 10 11 12 13 14 15   ········ ········
0040  16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25   ········ ·· !"#$%
```

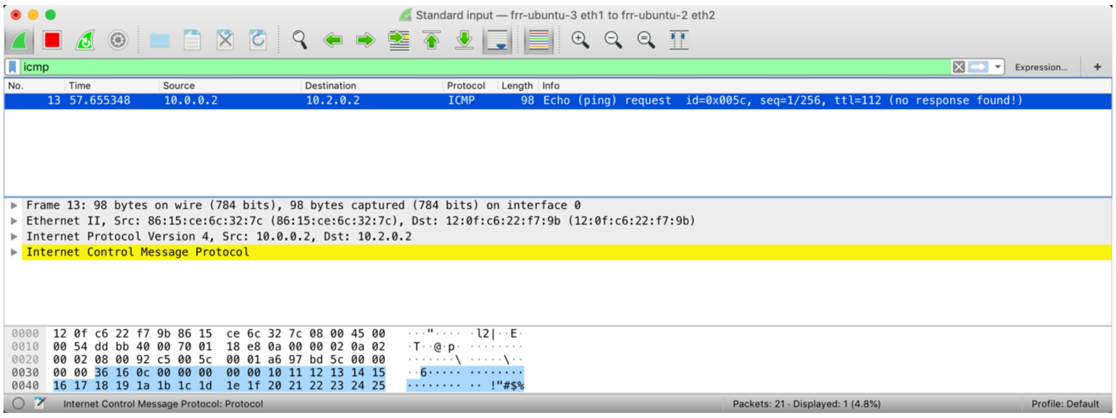
Internet Control Message Protocol: Protocol — Packets: 13 · Displayed: 1 (7.7%) — Profile: Default

Figure 3.7. Wireshark showing the unsuccessful ping through the secondary path between routers "frr-router-1" and "frr-router-3"

It has to be found why this problem happens. So, in order to get more information about this problem, it has to be repeated the ping, but this time with Wireshark listening, on the one hand, in the link between routers "frr-router-3" and "frr-router-2"; and on the other hand, in the link between routers "frr-router-2" and "ubuntu-2".

It can be observed in Figure 3.8 that between "frr-router-3" and "frr-ubuntu-2" there is the ping request. However, in Figure 3.9 it can be seen that the ping request never arrives to the link between "frr-router-2" and "ubuntu-2".

So, it can be concluded that "frr-router-2" is dropping the packet.



Standard input — frr-ubuntu-3 eth1 to frr-ubuntu-2 eth2

icmp

```
No.   Time        Source      Destination  Protocol  Length  Info
   13 57.655348   10.0.0.2    10.2.0.2     ICMP          98  Echo (ping) request  id=0x005c, seq=1/256, ttl=112 (no response found!)
```

```
▶ Frame 13: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
▶ Ethernet II, Src: 86:15:ce:6c:32:7c (86:15:ce:6c:32:7c), Dst: 12:0f:c6:22:f7:9b (12:0f:c6:22:f7:9b)
▶ Internet Protocol Version 4, Src: 10.0.0.2, Dst: 10.2.0.2
▶ Internet Control Message Protocol
```

```
0000  12 0f c6 22 f7 9b 86 15  ce 6c 32 7c 08 00 45 00   ···"···· ·l2|··E·
0010  00 54 dd bb 40 00 70 01  18 e8 0a 00 00 02 0a 02   ·T··@·p· ········
0020  00 02 08 00 92 c5 00 5c  00 01 a6 97 bd 5c 00 00   ·······\ ·····\··
0030  00 00 36 16 0c 00 00 00  00 00 10 11 12 13 14 15   ··6····· ········
0040  16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25   ········ ·· !"#$%
```

Internet Control Message Protocol: Protocol — Packets: 21 · Displayed: 1 (4.8%) — Profile: Default

Figure 3.8. Wireshark showing the unsuccessful ping through the secondary path between routers "frr-router-3" and "frr-router-2"

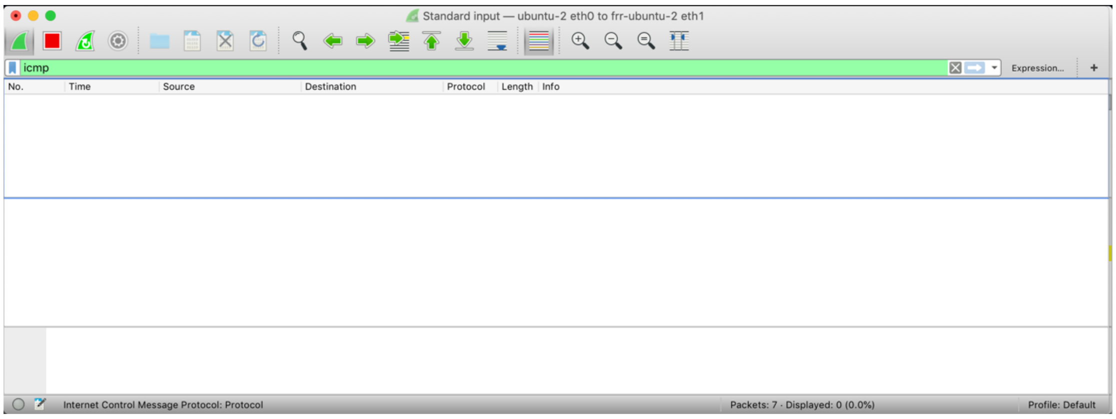

Standard input — ubuntu-2 eth0 to frr-ubuntu-2 eth1
icmp
Expression...
No.
Time
Source
Destination
Protocol
Length
Info
Internet Control Message Protocol: Protocol
Packets: 7 · Displayed: 0 (0.0%)
Profile: Default

Figure 3.9. Wireshark showing the unsuccessful ping through the secondary path between router "frr-router-2" and host "ubuntu-2"

The reason of this behaviour is that every router has activated reverse path filtering by default. When this functionality is activated, the routers expect to receive the packets through the shortest path which in this case is in the primary path. Therefore, when "frr-router-2" receives any packets through an interface that it does not expect, it drops the packets.

A solution is to deactivate the reverse path filtering in every interface setting to "0" the values that are in the "rp_filter" file:

> *sudo sysctl -w 'net.ipv4.conf.all.rp_filter=0'*
> *sudo sysctl -w 'net.ipv4.conf.default.rp_filter=0'*
> *sudo sysctl -w 'net.ipv4.conf.<INTERFACE NAME>.rp_filter=0'*

Once the reverse path filtering is deactivated, it can be checked again the behaviour of the routers when sending the ping with a TTL of 128.

Figure 3.10 shows the "ubuntu-1" console showing that the ping now has been successful. In Figure 3.11 it can be observed that the ping request goes through the secondary path as expected, but the response goes through the primary path. This happens because the default TTL value for "ubuntu-2" is 64 and thus, the packets follow the primary path. This is something to take into account when programming the code to do the measurements.

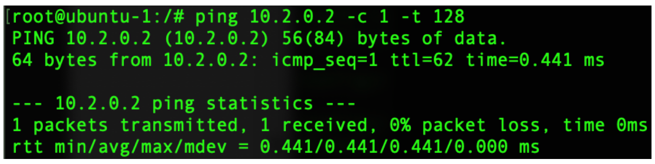
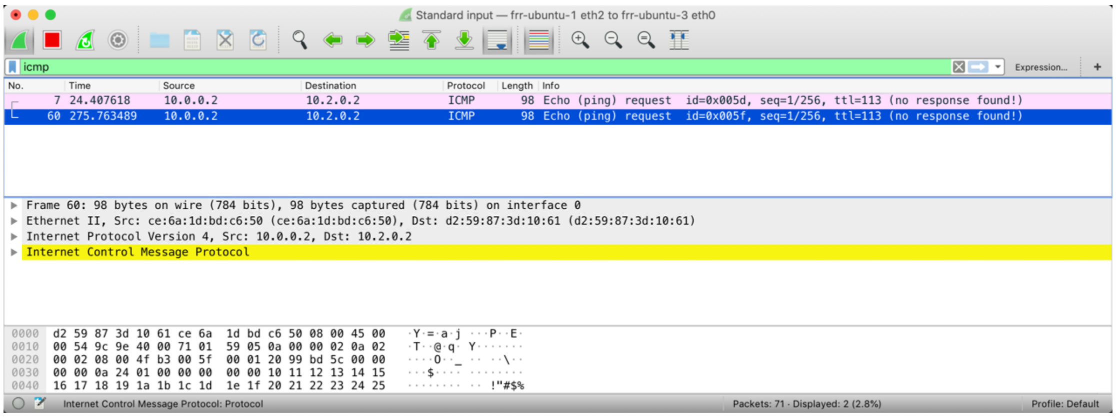
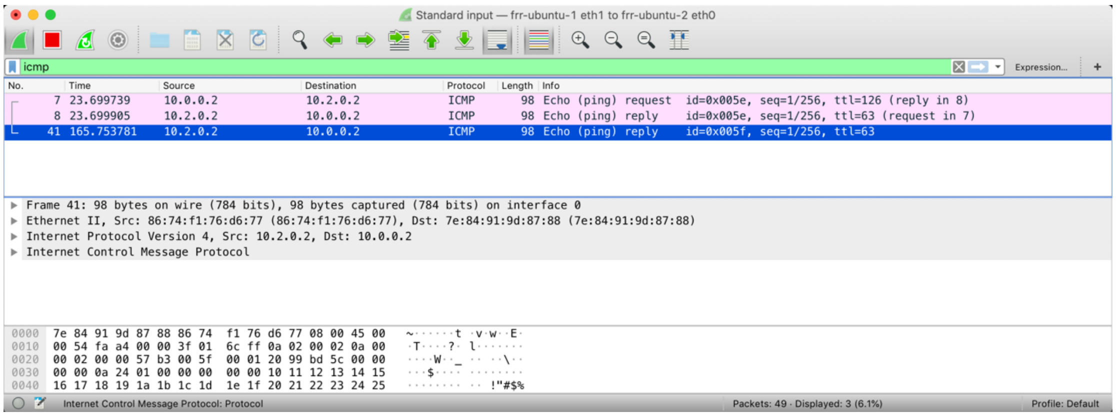

```
[root@ubuntu-1:/# ping 10.2.0.2 -c 1 -t 128
PING 10.2.0.2 (10.2.0.2) 56(84) bytes of data.
64 bytes from 10.2.0.2: icmp_seq=1 ttl=62 time=0.441 ms

--- 10.2.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.441/0.441/0.441/0.000 ms
```

Figure 3.10. "ubuntu-1" console with the successful ping through the secondary path



```
No.  Time        Source     Destination  Protocol  Length  Info
  7  24.407618   10.0.0.2   10.2.0.2     ICMP          98  Echo (ping) request  id=0x005d, seq=1/256, ttl=113 (no response found!)
 60  275.763489  10.0.0.2   10.2.0.2     ICMP          98  Echo (ping) request  id=0x005f, seq=1/256, ttl=113 (no response found!)

▶ Frame 60: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
▶ Ethernet II, Src: ce:6a:1d:bd:c6:50 (ce:6a:1d:bd:c6:50), Dst: d2:59:87:3d:10:61 (d2:59:87:3d:10:61)
▶ Internet Protocol Version 4, Src: 10.0.0.2, Dst: 10.2.0.2
▶ Internet Control Message Protocol

0000  d2 59 87 3d 10 61 ce 6a  1d bd c6 50 08 00 45 00   ·Y·=·a·j ···P··E·
0010  00 54 9c 9e 40 00 71 01  59 05 0a 00 00 02 0a 02   ·T··@·q· Y·······
0020  00 02 08 00 4f b3 00 5f  00 01 20 99 bd 5c 00 00   ····O··_ ·· ··\··
0030  00 00 0a 24 01 00 00 00  00 00 10 11 12 13 14 15   ···$···· ········
0040  16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25   ········ ·· !"#$%

Internet Control Message Protocol: Protocol     Packets: 71 · Displayed: 2 (2.8%)     Profile: Default
```

(a)



```
No.  Time        Source     Destination  Protocol  Length  Info
  7  23.699739   10.0.0.2   10.2.0.2     ICMP          98  Echo (ping) request  id=0x005e, seq=1/256, ttl=126 (reply in 8)
  8  23.699905   10.2.0.2   10.0.0.2     ICMP          98  Echo (ping) reply    id=0x005e, seq=1/256, ttl=63 (request in 7)
 41  165.753781  10.2.0.2   10.0.0.2     ICMP          98  Echo (ping) reply    id=0x005f, seq=1/256, ttl=63

▶ Frame 41: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
▶ Ethernet II, Src: 86:74:f1:76:d6:77 (86:74:f1:76:d6:77), Dst: 7e:84:91:9d:87:88 (7e:84:91:9d:87:88)
▶ Internet Protocol Version 4, Src: 10.2.0.2, Dst: 10.0.0.2
▶ Internet Control Message Protocol

0000  7e 84 91 9d 87 88 86 74  f1 76 d6 77 08 00 45 00   ~······t ·v·w··E·
0010  00 54 fa a4 00 00 3f 01  6c ff 0a 02 00 02 0a 00   ·T····?· l·······
0020  00 02 00 00 57 b3 00 5f  00 01 20 99 bd 5c 00 00   ····W··_ ·· ··\··
0030  00 00 0a 24 01 00 00 00  00 00 10 11 12 13 14 15   ···$···· ········
0040  16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25   ········ ·· !"#$%

Internet Control Message Protocol: Protocol     Packets: 49 · Displayed: 3 (6.1%)     Profile: Default
```

(b)

Figure 3.11. Wireshark showing the successful ping through the secondary path between routers "frr-router-1" and "frr-router-3" (a); and between routers "frr-ubuntu-1" and frr-ubuntu-2 (b)

3.3.1.3 Conclusions of first topology results

On the one hand, it has been verified that the multipath behaviour is possible and works the desired way. On the other hand, it has been observed that reverse path filtering has to be disabled to make the topology work properly. Furthermore, it has also been observed that the replies of the ping have always followed the primary path. This is something to take into account when programming the code for the measurements in order to put the correct forwarding rules.

3.4. TCP scripts

Once verified that the network behaves with the multipath capabilities, the script to do the measurements can be done. The idea is to check how multipath behaves compared to having a normal single path connection.

The task to be performed in the code are the following:

- Create a script which establish various TCP socket connections that follow different paths from the sender to the receiver
- Check failure resilience with the simple topology
- Create a normal TCP socket connection from the sender to the receiver

Final measurements will be performed in a more complex topology.

3.4.1 Multi-socket TCP script

As previously said, the aim is to programme a script that establish different socket connections that follow different from the sender to the receiver. For this purpose, each socket will be connected to the same IP address but to different port. The idea is to assign a different TTL value to each port so that each socket follows a different path.

It has to be done a code for the sender and another for the receiver with the corresponding mechanisms to behave the desired way and take the benefit of multipath.

3.4.1.1 Multi-socket-TCP sender script

I have programmed the code for the sender to send the data using the different paths at the same time.
The way of using the code through the console is by executing the following command:

```
sender_multi_tcp_threads.py <filename> <receiver_IP> <log_file_name>
```

The code starts by reading the name of the file that has to be sent to the receiver, as well as the IP address of the receiver. It also reads the name of a log file where the results will be stored.

Then it reads the data of the file and calculates the amount of data that has to be sent through each socket depending on the number of sockets:

$$data\_per\_socket = \frac{total\_data}{number\_of\_sockets}$$

Afterwards, the sockets and a thread per socket are created. The threads are created to send the data through each socket in parallel. Each socket calls the function "send_data" which has to receive a socket, the port of the socket, the number of the socket, and the beginning and the end of the data that has to be sent through the socket.

```
Thread(target = send_data, args = (socket_list[i], port_list[i], i, data_begin, data_end))
```

Where "socket_list" a python dictionary which contains the list of sockets created. "port_list" is a python dictionary which contains the list of port numbers related to the socket number. In this case "i" is the number of the socket that has just been created.

The “send_data” function first connects the socket to the receiver IP address and the corresponding port of that socket:

data_socket.connect((receiver_IP, port))

Then it sets the timeout of the socket to 1 second and the transmission begins. When the timeout is reached, the amount of data sent is checked.

If there has been data transmitted means that the socket is working properly. So, in that case, the variable that stores the amount of data that has already been sent through the socket is incremented in the amount of data sent during the last timeout. Furthermore, the value of the throughput in that moment is stored in a log file. This log file is important to analyse the final measurements.

When the total amount of data sent through the socket is equal to the amount of data that the socket had to transmit, then the socket stops transmitting the data and checks which of the sockets that still have not finish the transmission is the one with less throughput. Then it makes the port of that socket with less throughput to send the data with the TTL value of the socket that has just finished the transmission. By doing this, that socket will increment its throughput.

On the other hand, when the timeout is reached and no data has been sent means that there has been a failure in any link of the path that that socket follows. When this failure is detected, it is called the function “change_ttl” which assigns a different TTL value to the port of that socket. In this way, there is failure resilience.

When the transmission is finished, the socket is closed.

Figure 3.12 in next page shows the flow diagram of the code.

Appendix A shows the whole code for the sender.

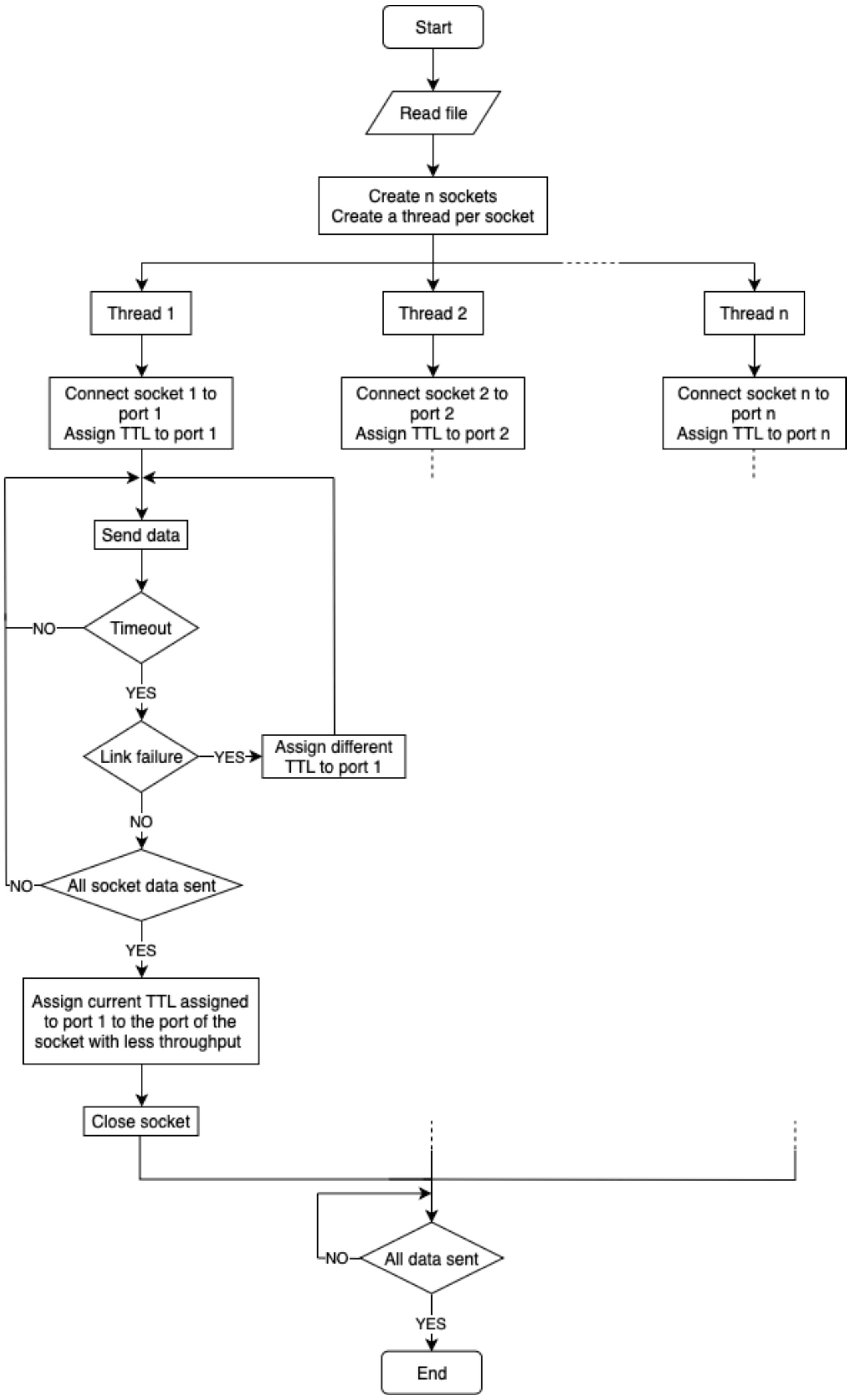

Start
Read file
Create n sockets
Create a thread per socket
Thread 1
Thread 2
Thread n
Connect socket 1 to port 1
Assign TTL to port 1
Connect socket 2 to port 2
Assign TTL to port 2
Connect socket n to port n
Assign TTL to port n
Send data
Timeout
NO
YES
Link failure
YES
Assign different TTL to port 1
NO
All socket data sent
NO
YES
Assign current TTL assigned to port 1 to the port of the socket with less throughput
Close socket
All data sent
NO
YES
End

Figure 3.12. Multi-TCP-sockets sender flow diagram

3.4.1.2 Multi-socket-TCP receiver script

Although the code for the sender and receiver present many similarities, they are different. The code for the receiver receives the data through different paths at the same time and has to process and reorganise it.

The way of using the code through the console is similar to the sender. The following command has to be executed:

receiver_multi_tcp_threads.py <filename> <receiver_IP>

In this case, the code starts reading the name that the user wants for the file received. It also reads the IP address of the receiver, that is, its own IP address.

Then it creates the sockets and binds them to the same ports as the sender:

socket_list[i].bind((receiver_IP, port_list[i]))

Where "socket_list" a python dictionary which contains the list of sockets created. "port_list" is a python dictionary which contains a list port numbers. These port numbers are the ones used for the connection with the sender. In this case "i" is the number of the socket that just has been created.

It is also created a temporal file for each socket. In this file each socket is going to store the data that they are receiving.

Then the threads are created to receive the data through each socket in parallel. In this case, each socket calls the function "receive_data" which has to receive as arguments just a socket and the number of that socket.

Thread(target = receive_data, args = (socket_list[i], i)))

The "receive_data" function first accepts the socket connection with the sender. When accepting this connection, it is received a new socket object which is the one that is going to be used to receive the data. It is also received the address bounded to the socket on the other end of the connection. This address has the IP address of the sender and the port number that the receiver is going to use to communicate with the sender:

conn, addr = data_socket.accept()

Where "conn" now stores this new socket and "addr" stores the address of the sender. Now that it is known the port that receiver has to use, the forwarding rules are assigned. In this way, to each of these ports is assigned a TTL value that is going to be used in the transmission.

Then the timeout is set to 1 second and the socket starts to receive the data:

data = conn.recv(PAYLOAD_SIZE)

When the socket receives an amount of data equal to "PAYLOAD_SIZE", this data is stored in the temporal file that corresponds to that socket. Furthermore, a variable that controls the amount of data received is incremented in the total amount of data received. This variable is useful when we want to change the TTL value assigned to the port of the socket with worse throughput. Then the socket continues receiving data.

When the timeout is reached, it is checked if there has occurred an exception or not. If there is no exception, that means that the transmission is being carried out properly and the socket continues receiving data.

On the contrary, if there is an exception, it means that there has been a failure in any link of the path that the socket follows. In this case, it is called the function "change_ttl" and a different TTL value to the port of that socket is assigned. Therefore, there is failure resilience.

When there is no more data to receive, the socket stops receiving data and checks which of the sockets that still have not finish the data reception is the one with less throughput. Then it makes the port of that socket with less throughput to send the data with the TTL value of the socket that has just finished the transmission, and thus, the throughput of that socket is incremented. And afterwards, the socket is closed.

When all the sockets have finished the data reception and stored the data they have receive in their corresponding temporal files, it is called the function "check_to_write()" which reads the data from the different temporal files in order (first the data of the file used by the socket 1, then the one used by the socket 2…) and stores the data in the final received file which is an exact copy of the file that the sender has sent.

Figure 3.13 in next page shows the flow diagram of the code.

Appendix B shows the whole code for the receiver.

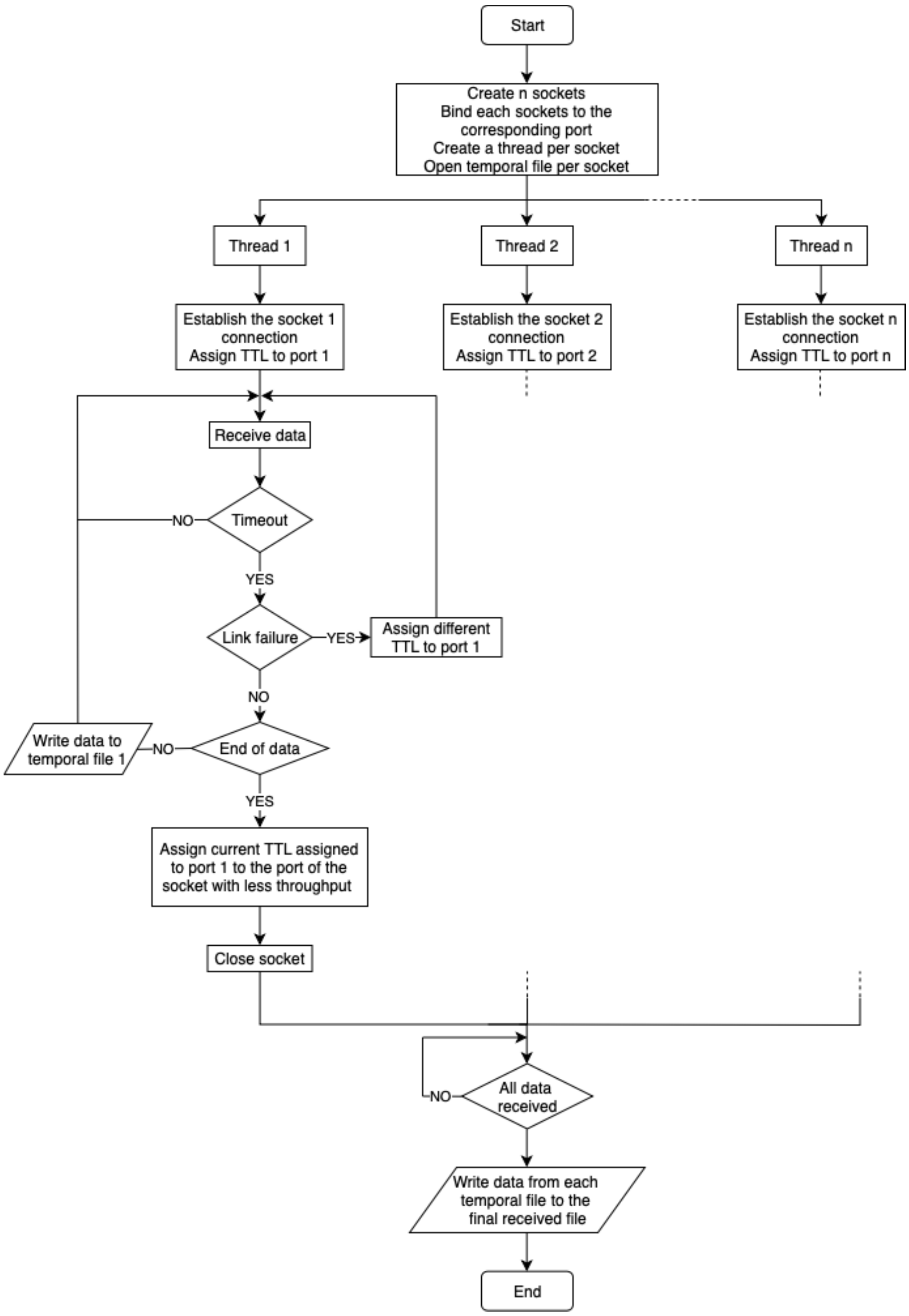

Start
Create n sockets
Bind each sockets to the corresponding port
Create a thread per socket
Open temporal file per socket
Thread 1
Thread 2
Thread n
Establish the socket 1 connection
Assign TTL to port 1
Establish the socket 2 connection
Assign TTL to port 2
Establish the socket n connection
Assign TTL to port n
Receive data
Timeout
NO
YES
Link failure
YES
Assign different TTL to port 1
NO
End of data
NO
Write data to temporal file 1
YES
Assign current TTL assigned to port 1 to the port of the socket with less throughput
Close socket
All data received
NO
Write data from each temporal file to the final received file
End

Figure 3.13. Multi-TCP-sockets receiver flow diagram

3.4.2 Single-socket-TCP script

In order to do the comparison between the behaviour of multipath and single path, it is also necessary a script that uses a single path. This script has to open a normal TCP socket connection and send all the data through that socket.

Unlike the sockets in the multipath script, the socket in this script does not change the TTL and thus, it is always transmitting through the primary path.

3.4.2.1 Single-socket-TCP sender script

The way of using the code through the console is very similar as the one used in the multi-socket scripts. It has to be executed the following command:

sender_single_tcp.py <filename> <receiver_IP> <port_number>

The code starts by reading the name of the file that has to be sent to the receiver, as well as the IP address of the receiver and the port number for the connection.

Then it connects the socket to the receiver IP address and the port:

data_socket.connect((receiver_IP, port))

Afterwards, it reads the data of the file, sets the timeout to 1 second and starts transmitting. This timeout is useful to have a control of the data that is being sent every second.

When all the data is transmitted, the socket closes and the script ends.

3.4.2.2 Single-socket-TCP receiver script

In this case, the way of executing the code is similar to the one used for the sender:

receiver_single_tcp.py <filename> <receiver_IP> <port_number>

The code starts by reading the name of the file that is going to be received, as well as the IP address of the receiver and the port number for the connection. This IP address is its own address since it is the receiver.

Then it binds the socket to its IP address and the port that the user has indicated.

data_socekt.bind((receiver_IP, receiver_port))

Afterwards, it accepts the socket connection with the sender. When this connection is accepted, it receives a new socket object which is the one that is going to be used to receive the date. It also receives the address bounded to the socket on the other end of the connection. This address contains the IP address of the sender, as well as the port number that the receiver needs for the communication with the sender:

conn, addr = data_socket.accept()

Where "conn" is the new socket and "addr" is the address of the sender. Then, the socket starts to receive data.

The socket is continuously receiving data and writing it in the output file. It receives data until the end of the data is reached. Then, the socket closes and the script ends.

3.5. Abilene topology

Once the scripts are created and the routers work the desired way, it is finally the time to take the measurements from a more complex network in order to have more realistic outcomes. It is going to do the measurements for the three different the use cases.

The topology chosen to do these measurements is the Abilene topology which can be seen in Figure 3.14. This topology is interesting because it has many links and many possible paths.

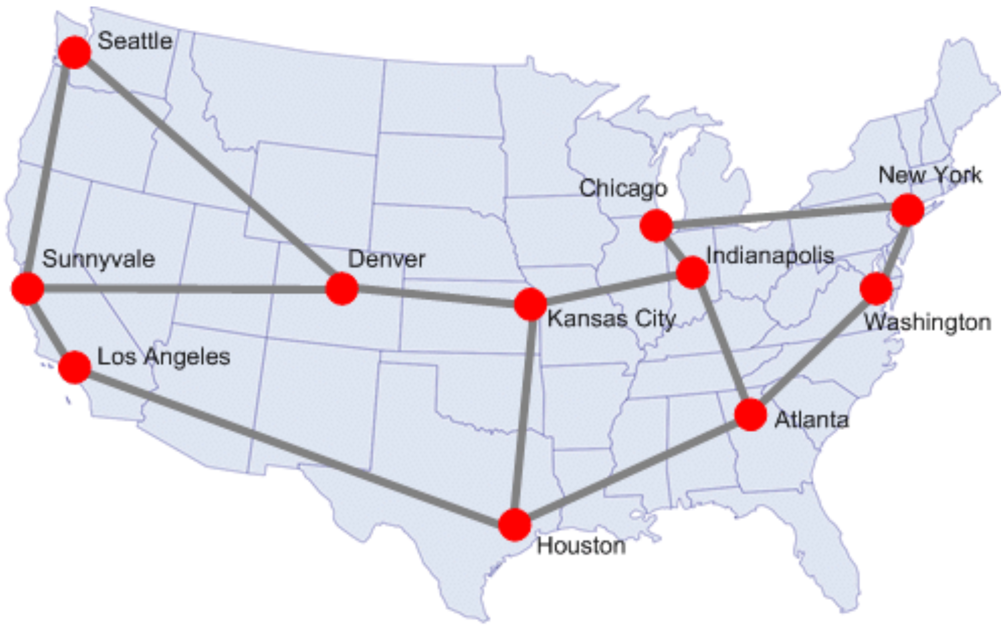


Figure 3.14. Abilene topology [1]

Therefore, the next step is to build this topology in GNS3 and perform the measurements. Figure 3.15 shows this topology in GNS3. The speed of each link is 10Mb/s. Next to the links it can be seen a number which represents it cost. This cost numbers have been taken from [1]. The costs are used by the OSPF algorithm to calculate the routes and store them in the routers. In this case, I added manually the cost values into the OSPF daemon configuration file in order to have a topology similar to the one implemented in [1].

In order to perform the measurements, in the topology it has been added 2 hosts: “ubuntu-1” and “ubuntu-2”. “ubuntu-1” is connected to router “frr-ubuntu-1”, whereas “ubuntu-2” is connected to router “frr-ubuntu-6”. It has been chosen these routers to connect the hosts in order to have multiple paths between them. The measurements are going to be carried out from host “ubuntu-1” to host “ubuntu-2”.

As commented in Chapter 2, these measurements are going to be performed to study the behaviour of the network for the following 3 use cases:

1. Normal transmission
2. Transmission limiting the bandwidth of some links of the shortest path
3. Transmission deleting a link of the shortest path

For each use case, it will be measured the throughput for a normal connection with one TCP socket that follows the shortest path, and also for a connection with various TCP sockets which follow different paths.

The first use case will show which method is better in ideal conditions where there are not any interruptions when transmitting, nor other transmission at the same time. This use case is studied in Section 4.1.1.

The second use case will check if multipath behaves better than single path when having the shortest path overloaded. This use case is studied in Section 4.1.2.

The third use case will check if multipath can detect a link failure and uses an alternative path to send the data, and thus, not having interruptions in the transmission. This use case is studied in Section 4.1.3.

In Appendix C all the different paths from “ubuntu-1” to “ubuntu-2” are shown for each TTL value.

In Appendix D all the different paths from “ubuntu-2” to “ubuntu-1” are shown for each TTL value.

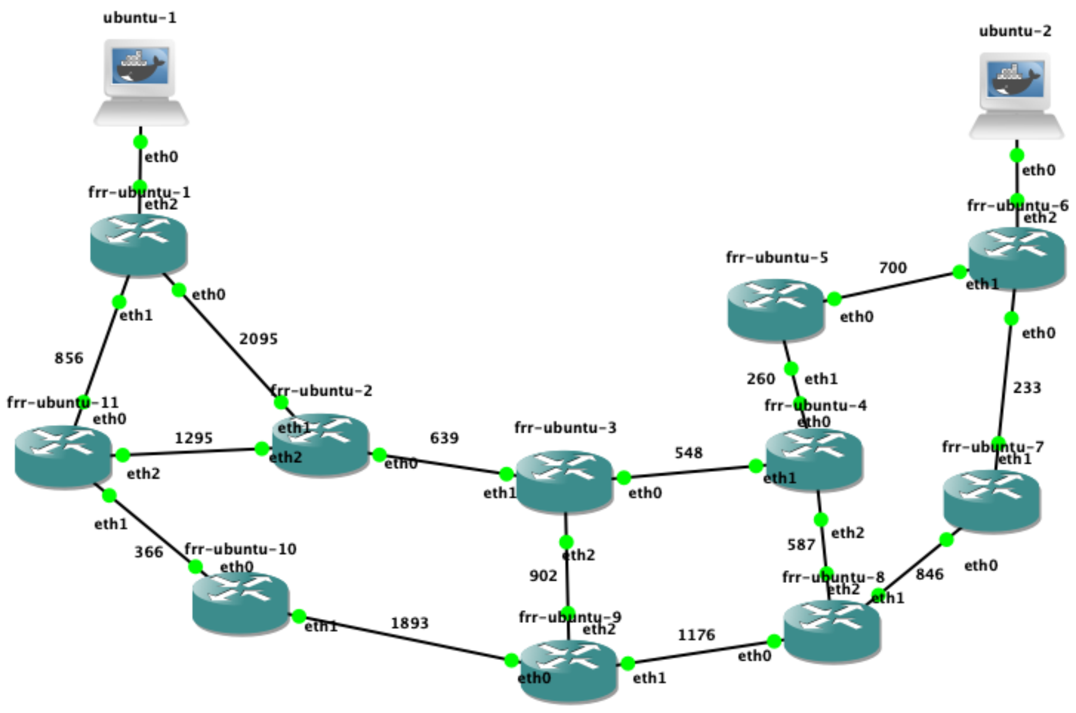


Figure 3.15. Abilene topology in GNS3

Chapter 4.

Results

This chapter is divided into two sections. First section shows the results of the measurements made for the different use cases.

Second section shows the issues that I found out while carrying out the measurements related to the paths.

4.1. Use cases results

Next sections show the results of the measurements for the different use cases. The results are measured by sending the same file. It is measured the transmission time that it takes for sending the whole file, as well as the bytes that have been sent until a certain time. Therefore, the curves that show these bytes sent, grow until the amount of bytes sent is equal to the length of the file transmitted.

These measurements have been done for 4 sockets. I considered 4 different paths enough to study the multipath behaviour in the above topology.

The transmission is performed from host "ubuntu-1" to host ubuntu-2" (see Figure 3.15).

The TTL value of each socket has been chosen after knowing which TTL follows each path. In a real application this is not possible, so one the further improves of the code programmed could be to make the application to check the different TTL possibilities and afterwards do the TTL election. But for this study this is not necessary.

The measurements will be carried out for the transmission of 2 different files in order to have more outcomes to analyse. The size of each file transmitted are shown in Table 4.1.

File Name	**Size**
sample-video.mp4	514,832,018 bytes (490.98 Megabytes)
sample-video2.avi	220,514,438 bytes (210.30 Megabytes)

Table 4.1. Transmitted files size

4.1.1 Use Case 1: Ideal Transmission

The first use case is a transmission in ideal conditions where there are not any interruptions when transmitting, nor other transmission at the same time. Therefore, the transmission can take the whole bandwidth.

4.1.1.1 Bigger file transmission in use case 1

In this section are shown the results when transmitting the file “sample-video.mp4” which has a size of 514,832,018 bytes (490.98 Megabytes).

Method	**Transmission time**
Multipath	55.56 seconds
Single path	45.1 seconds

Table 4.2. Use Case 1. Bigger file transmission time

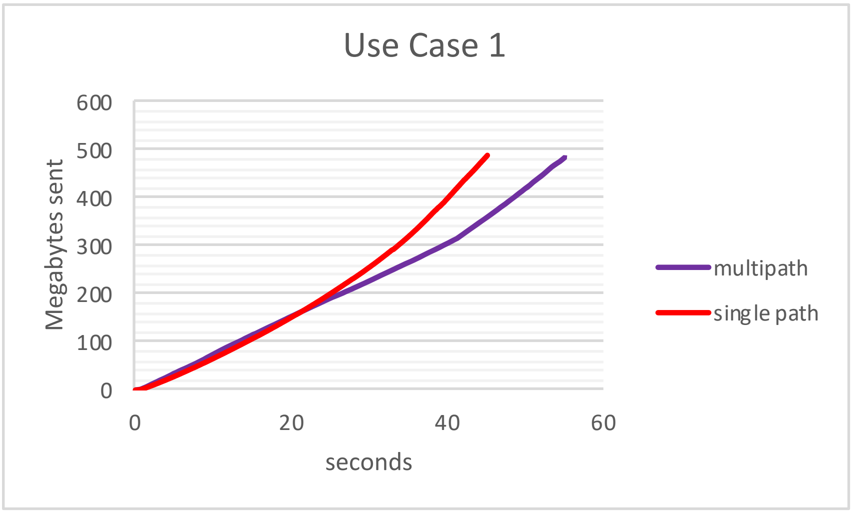


Figure 4.1. Use Case 1. Comparison of the bigger file transmitted using multipath and single path

Table 4.2 shows the total transmission time needed to send the bigger file from host “ubuntu-1” to host “ubuntu-2”.

Figure 4.1 shows the graphic comparing the bytes sent in both methods.

It can be observed in the outcomes of this use case that when using single path to transmit the file the transmission time is a bit smaller than using multipath and hence, its throughput is better.

4.1.1.2 Smaller file transmission in use case 1

In this section are shown the results when transmitting the file “sample-video-2.avi” which has a size of 220,514,438 bytes (210.30 Megabytes).

Method	**Transmission time**
Multipath	21.52 seconds
Single path	16.05 seconds

Table 4.3. Use Case 1. Smaller file transmission time

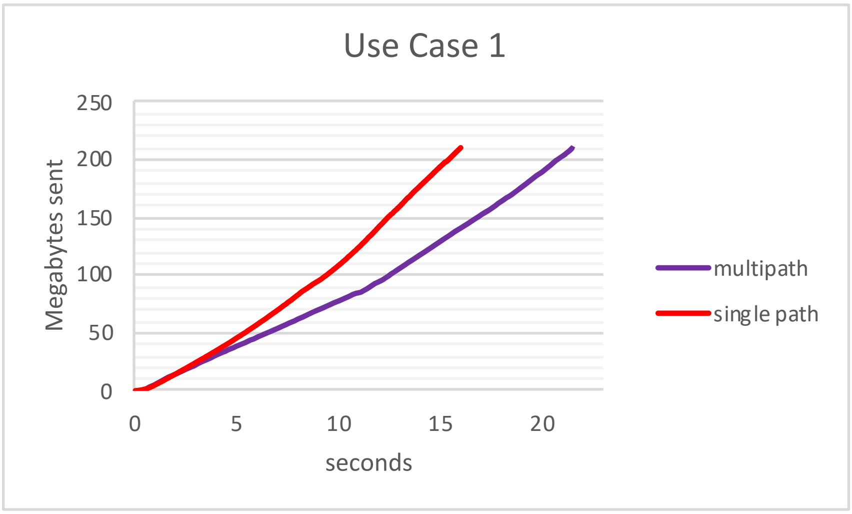


Figure 4.2. Use Case 1. Comparison of the smaller file transmitted using multipath and single path

Table 4.3 shows the total transmission time needed to send the smaller file from host “ubuntu-1” to host “ubuntu-2”.

Figure 4.2 shows the graphic comparing the bytes sent in both methods.

It can be observed the same as when transmitting the bigger file: when using single path to transmit the file, the transmission time is a bit smaller than using multipath and hence, its throughput is better.

4.1.1.3 Use Case 1 conclusions

It can be observed that, in this use case, when transmitting whatever of the two files, single path transmits the file faster and hence, it offers a better throughput. The reason of this behaviour is that the multipath code introduces some extra processes that the host has to handle and thus, making the transmission slower.

This outcome may be improved by improving the code.

4.1.2 Use Case 2: Reduced bandwidth transmission

The way decided to measure this use case is opening an additional transmission between routers “frr-ubuntu-3” and “frr-ubuntu-5” (see Figure 3.15) that belong to the shortest path. This makes the two links between these routers to be overloaded. Figure 4.3 shows the links that are overloaded (represented in red colour) when this new transmission is opened between the mentioned routers.

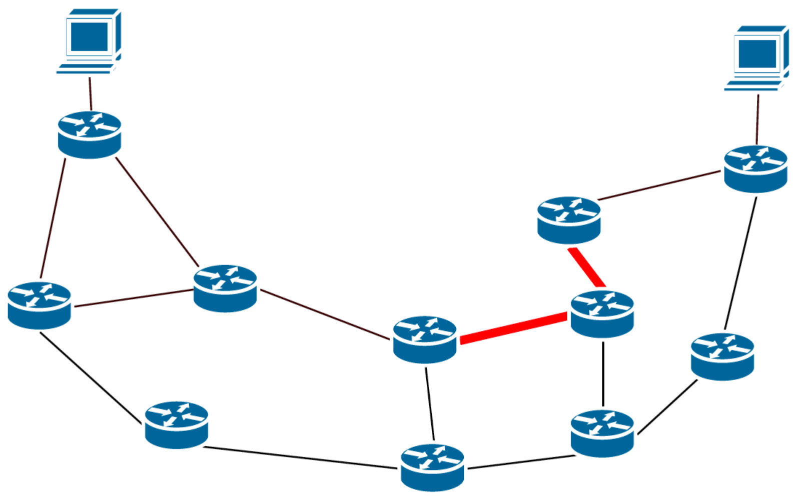


Figure 4.3. Overloaded links in Use Case 2

A traditional routing protocol calculates the shortest path and stores it in the routers and thus, all the transmission from one sender to one receiver are performed through that path. As this path is supposed to be the best, the probability of that path to be overloaded is high.

In this use case it is measured what happens when transmitting through an overloaded path and compare which method is better: multipath or single path.

4.1.2.1 Bigger file transmission in use case 2

In this section are shown the results for use case 2 when transmitting the file “sample-video.mp4” which has a size of 514,832,018 bytes (490.98 Megabytes).

Method	**Transmission time**
Multipath	67.79 seconds
Single path	239.26 seconds

Table 4.4. Use Case 2. Bigger file transmission time

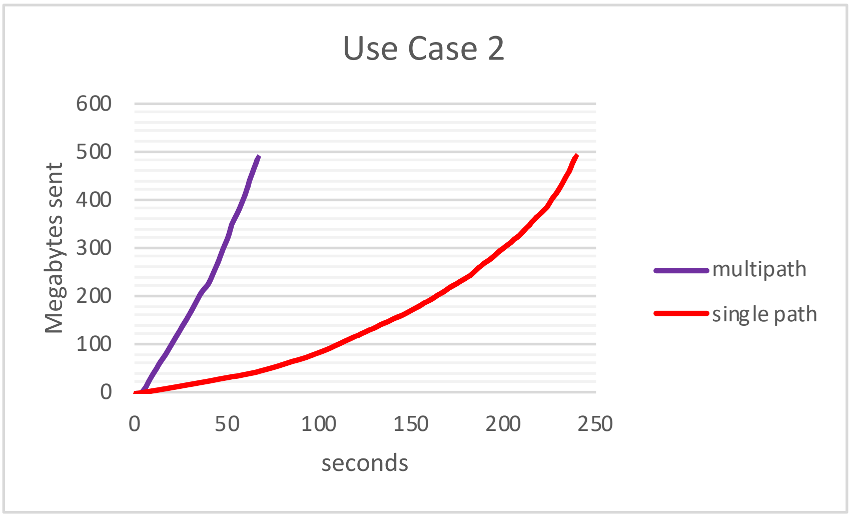


Figure 4.4. Use Case 2. Comparison of the bigger file transmitted using multipath and single path

Table 4.4 shows the total transmission time needed to send the bigger file from host "ubuntu-1" to host "ubuntu-2".

Figure 4.4 shows the graphic comparing the bytes sent in both methods.

It can be clearly observed that the transmission time when transmitting by using multipath is more or less the same as in the previous use case; whereas when transmitting by using single path the delay of the transmission is very high compared to the previous use case.

4.1.2.2 Smaller file transmission in use case 2

In this section are shown the results of use case 2 when transmitting the file "sample-video-2.avi" which has a size of 220,514,438 bytes (210.30 Megabytes).

Method	**Transmission time**
Multipath	22.05 seconds
Single path	83.81 seconds

Table 4.5. Use Case 2. Smaller file transmission time

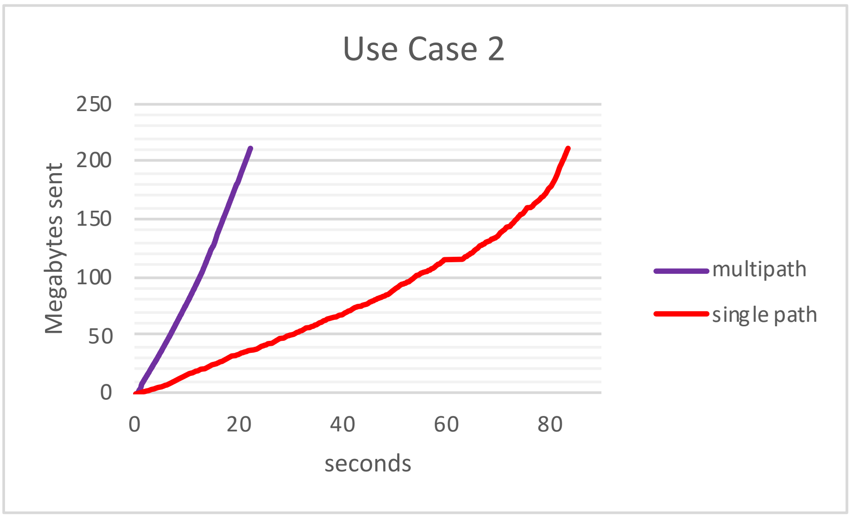


Figure 4.5. Use Case 2. Comparison of the smaller file transmitted using multipath and single path

Table 4.5 shows the total transmission time needed to send the smaller file from host “ubuntu-1” to host “ubuntu-2”.

Figure 4.4 shows the graphic comparing the bytes sent in both methods.

The behaviour of multipath compared to single path when transmitting the smaller file in this use case is exactly the same as when transmitting the bigger file: it can be clearly observed that the transmission time when transmitting by using multipath is more or less the same as in the previous use case; whereas when transmitting by using single path the delay of the transmission is very high compared to the previous use case.

4.1.2.3 Use Case 2 conclusions

Although the shortest path is supposed to be the best and the fastest, this assumption does not take into account that this path can be overloaded and could have more delays than other paths.

For this reason, it can be seen in previous outcomes for use case 2 that the throughput in this use case when transmitting whatever of the two files is much better for multipath and the file is transmitted faster. By using multipath, the end-users have the advantage of deciding which path to use. When they experience that a path is slower than others, the path can be changed by just changing the TTL value.

4.1.3 Use Case 3: Removing a link during transmission

The last measurement is performed by removing a link while the connection is being carried out. The link that is going to be removed during the transmission is the one between “frr-ubuntu-4” and “frr-ubuntu-5” (see Figure 3.15).

In this use case it has been considered a static topology where there are no routes recomputation when there is any change in the state of the topology like a change in the state of a link.

This assumption has its baseline in the "*Dynamic Route Computation Considered Harmful*" paper [2] which was explained in Section 2.1.2. In that paper, the authors discuss that the route recomputation has an important impact in the topology throughput. Besides, they assume that topology changes are far less frequent than status changes. This assumption is supported by a study done in [6] where failures with time-to-repair longer than 24 hours were 3.7% of all failures.

Furthermore, in the static topology, the effect of the failure resilience will be shown more clearly for the multipath case.

In the other use cases this assumption was not necessary since there were no changes in the topology and thus, neither in the link states.

Figure 4.6 shows the topology with a red cross in the link that is going to be removed during the transmission.

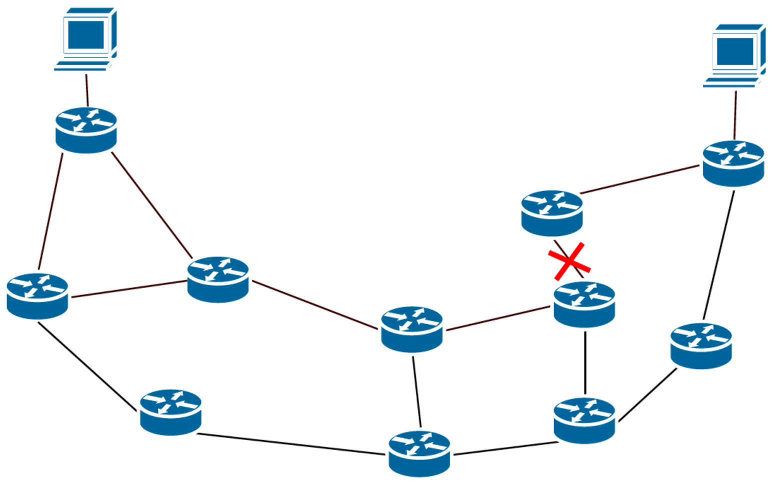


Figure 4.6. Link to be removed in Use Case 3

4.1.3.1 Bigger file transmission in use case 3

In this section are shown the results for use case 3 when transmitting the file "sample-video.mp4" which has a size of 514,832,018 bytes (490.98 Megabytes).

The link is removed when the transmission is in the second 20.

Method	**Transmission time**
Multipath	58.58 seconds
Single path	-

Table 4.6. Use Case 3. Bigger file transmission time

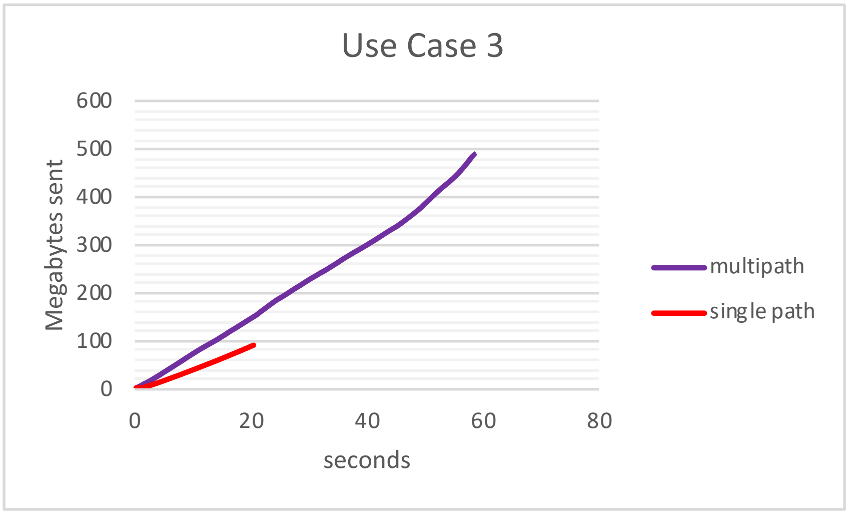


Figure 4.7. Use Case 3. Comparison of the file bigger transmitted using multipath and single path

Table 4.6 shows the total transmission time needed to send the smaller file from host “ubuntu-1” to host “ubuntu-2”.

Figure 4.7 shows the graphic comparing the bytes sent in both methods.

It can be observed in these outcomes that in the multipath case, the transmission continues and last more or less the same as the previous use cases; whereas for the single path case, the transmission is stopped when the link is removed.

4.1.3.2 Smaller file transmission in use case 3

In this section are shown the results in use case 3 when transmitting the file “sample-video-2.avi” which has a size of 220,514,438 bytes (210.30 Megabytes).

The link is removed when the transmission is in the second 7.

Method	**Transmission time**
Multipath	19.56 seconds
Single path	-

Table 4.7. Use Case 3. Smaller file transmission time

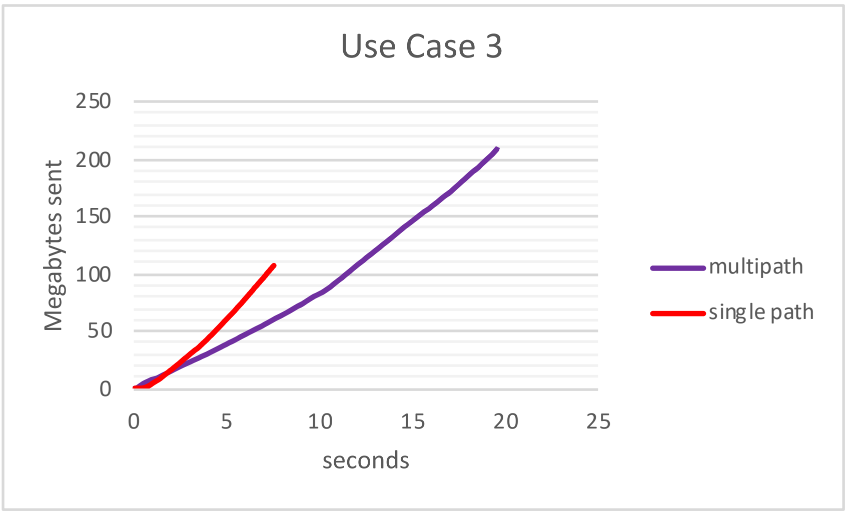


Figure 4.8. Use Case 3. Comparison of the smaller file transmitted using multipath and single path

Table 4.7 shows the total transmission time needed to send the smaller file from host “ubuntu-1” to host “ubuntu-2”.

Figure 4.8 shows the graphic comparing the bytes sent in both methods.

It can be observed that in this use case, when transmitting the smaller file, happens the same as when transmitting the bigger file: in the multipath case, the transmission continues and last more or less the same as the previous use cases; whereas for the single path case, the transmission is stopped when the link is removed.

4.1.3.3 Use Case 3 conclusions

It can be seen in previous outcomes for the use case 3 that when using multipath, the transmission continues even when there is a failure in a link. On the contrary, the single path transmission is dropped when there is a failure in one of the links of the shortest path.

Hence, using multipath has not cuts in the transmission when there are other possible paths to continue transmitting the data.

4.1.4 Comparison of multipath in the 3 use cases

Figure 4.9 shows a comparison of the measurements shown previously but only in the multipath case when transmitting the bigger file, whereas Figure 4.10 shows the comparison of the measurements when transmitting the smaller file. It is interesting to compare the behaviour of multipath in the different use cases in order to check if the load balancing or link failure affect the throughput of the transmission.

Although it can be observed that the worst result, when transmitting whatever of the two files, was for the use case 2 where the primary path was overloaded, this difference is not very significant and hence, it can be said that multipath is not affected by the state of the links in the network since it can change the path to transmit the data.

Therefore, it can be seen that multipath can adapt to the state of the network in that moment and transmit through a different path when a path is overloaded or there has been a failure.

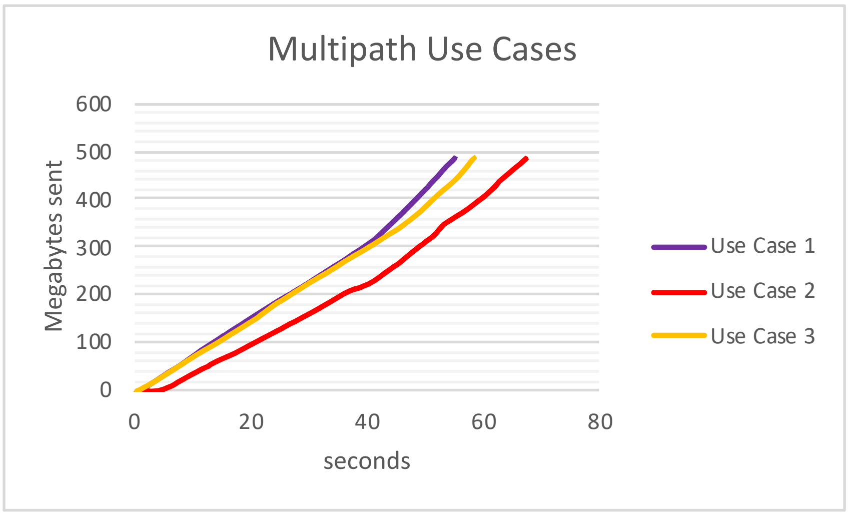


Figure 4.9. Comparison of multipath in the 3 use cases when transmitting the bigger file

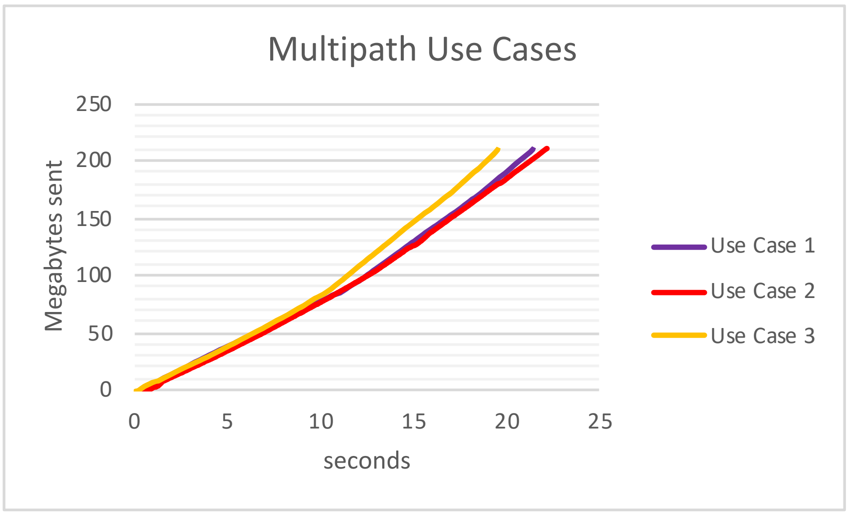


Figure 4.10. Comparison of multipath in the 3 use cases when transmitting the smaller file

4.2. Issues with the routing algorithm

When performing the measurements in the Abilene topology, I found out some unexpected issues related to the routing algorithm and the forwarding rules.

I experienced that certain TTL values produced loops in the path the packets follow from the sender to the receiver, more often than it was expected in the theoretical algorithm design. The loops were not infinite loops, so the transmission could continue. However, these loops result in an unexpected behaviour that has some impact in the transmission time, and thus, the throughput.

Although all the loops are very similar and have more or less the same impact in the throughput, they could be classified into 3 different groups:

- **Secondary – Primary loops:** loops that happen when a router deflects a packet through its secondary next hop and the following router forwards the packet through its primary next hop which is the previous router.

- **Secondary – Secondary loops:** loops that happen when a router deflects a packet through its secondary next hop and the following router also deflects the packet through its secondary next hop which is the previous router.

- **Primary – Secondary – Primary loops:** these loops are the worst I found since they traverse the same link three times. They happen when a router deflects a packet through its secondary next hop which is the previous router, and hence, this router has to send again the packet through its primary path returning the packet for second time to the same router.

In next sections it is analysed the probability of having these loops when transmitting from host "ubuntu-1" to host "ubuntu-2" and vice versa.

4.2.1 Loops from host "ubuntu-1" to host "ubuntu-2"

Table 4.8 shows the number of TTL values that experience each kind of loop and the total number of TTLs that are considered not a good choice for the transmission. Furthermore, it is shown which TTL value experiences each type of loop.

The TTL values that have been considered for this research are the ones from 127 to 255. The reason is that the interesting thing about multipath is having different good paths. Packets with TTL value equal or less 127 will always follow the primary path. Therefore, a good TTL value for the primary path is one equal or less 127 (there are other values higher than 127 that also follow the primary path), and the other TTL chosen should follow a different path and thus, values higher 127 have to be chosen.

Kind of loop	**Number of TTLs**	**TTL value**
Secondary - Primary	3	129, 144, 145
Secondary - Secondary	14	146, 147, 162, 163, 178, 179, 194, 195, 210, 211, 226, 227, 242, 243
Primary - Secondary - Primary	7	132, 148, 164, 196, 212, 228, 244
Total	**24**	

Table 4.8. Types of loops from host "ubuntu-1" to host "ubuntu-2"

As the considered TTL values are those from 127 to 255, there are 129 possible values to choose. Therefore, the percentage of choosing a wrong value is the amount of wrong values divided by the total number of values:

$$\frac{Wrong\_TLLs}{Total\_TTLs} = \frac{24}{129} \cong 19\%$$

Hence, the possibility of choosing a wrong TTL is 19%.

In order to explain this behaviour clearer, following figures show each kind of loop. These figures show the path that is followed when selecting the TTL value 129, 146 and 132. When the router deflects the packet through the secondary next hop, the link is displayed in blue colour, whereas if the router forwards the packet through the primary next hop, the link is displayed in red colour. When the router is about to deflect the packet though the secondary next hop, it decreases the TTL value of the packet with a constant determined by the IMRE routing algorithm. This decrement is also shown in the figures inside a rectangle. It is also shown the current TTL in each hop at the end of the arrow of the links.

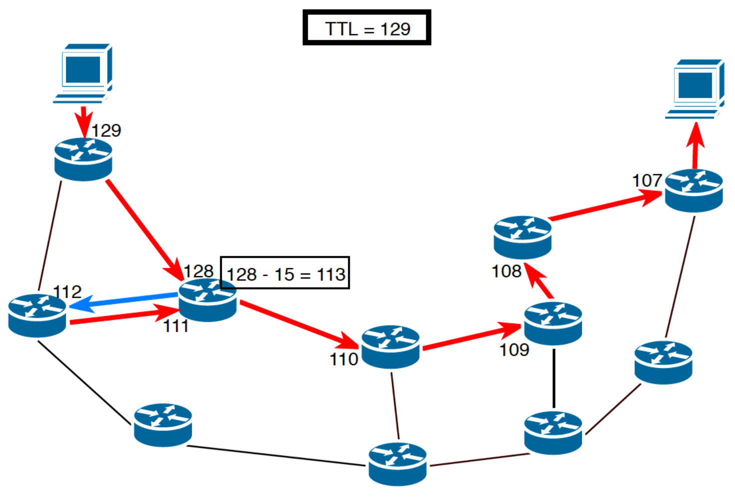


Figure 4.11. Secondary – Primary loop

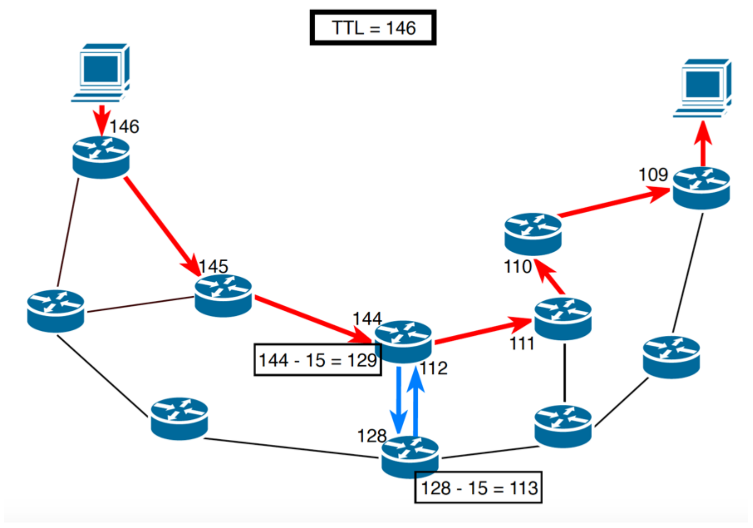


Figure 4.12. Secondary – Secondary loop

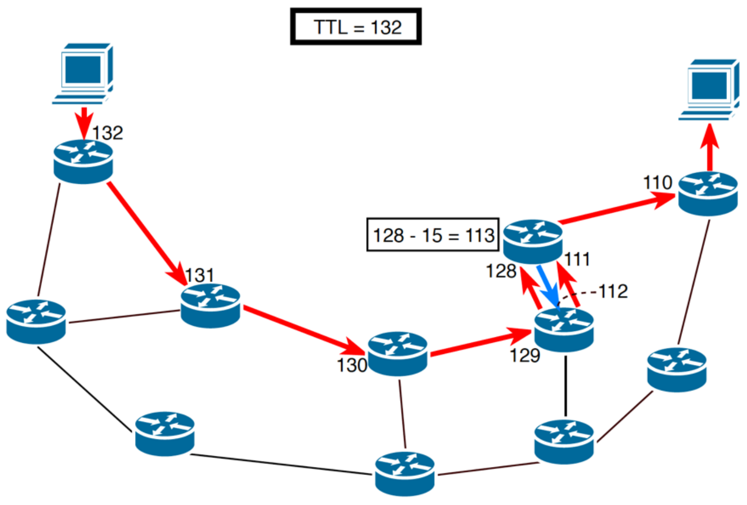


Figure 4.13. Primary – Secondary – Primary loop

4.2.2 Loops from host “ubuntu-2” to host “ubuntu-1”

Table 4.9 shows the number of TTL values that experience each kind of loop and the total number of TTLs that are considered not a good choice for the transmission. Furthermore, it is shown which TTL value experiences each type of loop. Again, the TTL values that have been considered for this research are the ones from 127 to 255.

Kind of loop	**Number of TTLs**	**TTL value**
Secondary - Primary	1	128
Secondary - Secondary	8	130, 147, 163, 179, 195, 211, 227, 243
Primary - Secondary - Primary	8	129, 145, 161, 177, 193, 209, 225, 241
Total	**17**	

Table 4.9. Types of loops from host “ubuntu-2” to host “ubuntu-1”

The percentage of choosing a wrong value is the amount of wrong values divided by the total number of values:

$$\frac{Wrong\_TLLs}{Total\_TTLs} = \frac{17}{129} \cong 13\%$$

Hence, the possibility of choosing a wrong TTL is the 13%.

Chapter 5.

Conclusions

The aim of the project was to build a multipath routing architecture able to transmit through different paths while remaining compatible to single path.

Multipath offers many advantages such as failure resilience, load balancing, and better throughput. However, it is not widely deployed.

This project contributes to verify all these benefits while keeping the backwards capability.

Throughout this project, it has been verified that the architecture designed is able to perform multipath routing by sending the packets through the best paths and thus, making use of the whole capacity of the network.

5.1. Conclusions about the measurements

The measurements made demonstrate that multipath is able to adapt to the network requirements and transmit a file in a more efficient manner.

Although in the measurements of an ideal transmission single path has had a better throughput, this difference was not very significant and can be associated to the extra processes that the multipath code has to deal with.

Furthermore, it has been demonstrated that when the primary path is overloaded, multipath can adapt to this circumstance and transmit through the better path in that moment. In this case, the throughput using multipath was much higher than using single path.

Finally, it has been also verified that when a link failure occurs, multipath continues the transmission without any penalty in the throughput by just changing the path, whereas TCP is not able to continue transmitting. This behaviour has been measured in a static topology where there are no routes recomputation when the topology changes.

5.2. Conclusions about the loops in the paths

When building the topology and checking the paths, it has been experienced some unexpected loops in some paths when using certain TTL values. The probability of choosing a bad path is also calculated. However, this calculated value cannot be taken as a general rule, since more measurements in different topologies and between different end-devices should be performed in order to have a more general rule about this loop behaviour.

These loops are related to the forwarding rules the IMRE algorithm used to calculate the secondary next hops. Although these loops appear in some TTL values, it has been demonstrated that this is a good algorithm which offers several good paths to reach a destination.

This loop-behaviour could be avoided by making the application check the different TTLs before the transmission. In order to keep full control over the network and speed up the convergence process, the network operator can also provide a set of good TTL values to the end hosts, which can be used for data transmission.

It has also been verified that the forwarding rule used offers many good different paths despite the loops. It might also be avoided by choosing a different forwarding rule.

5.3. Future work

As previously said, throughout this project it has been demonstrated that it is possible to build a multipath architecture while maintaining backward compatibility. Therefore, this project contributes to demonstrate the benefits when using multipath.

However, the multi-socket-TCP code programmed can be improved in various aspects:

- Selecting the TTL values dynamically instead of using pre-configured values.
- The way of changing the TTL when a path is worse than others (overloaded path) can be improved in order to detect earlier that a path is slower than others.
- The best way I found to send the packets through the different sockets at the same time was using threads. However, there are other ways to do this and maybe it could be programmed in a more efficient way.

Concluding, the code may be improved to work more efficiently.

On the other hand, this project has also contributed to check the IMRE algorithm. It has been demonstrated that this algorithm and the forwarding rules offer different good paths between end-users. However, it has been experienced that when using certain TTL values, there are some loops in the path. Therefore, a future work is to keep studying this algorithm and if different forwarding rules offer better behaviour.

A dynamic selection of the TTL values after checking the good values could also solve the problem of the possibility of choosing a wrong TTL value.

Appendix A. Multi-socket-TCP sender code

```
# sender_multi_tcp_threads.py

import socket                  # Import socket module
import time
import sys
import math
import os
from threading import Thread

NUMBER_OF_SOCKETS = 4

def change_ttl(socket_number):

    global port_list
    global ttl_list
    global ttl_socket

    count = 0

    i = socket_number + 1

    if i == NUMBER_OF_SOCKETS:
        i = 0

    ttl_temp = ttl_socket[socket_number]

    while count < NUMBER_OF_SOCKETS - 1:

        if ttl_socket[i] is not None:
            ttl_socket[socket_number] = ttl_socket[i]
            break

        i = i + 1

        if i == NUMBER_OF_SOCKETS:
            i = 0
        count = count + 1

    if ttl_socket[socket_number] == None or ttl_temp == ttl_socket[socket_number]:
        for i in range (0, NUMBER_OF_SOCKETS):
            ttl_socket[i] = ttl_list[i]

    os.system('iptables -t mangle -A POSTROUTING -p TCP --tcp-flags ACK ACK --
dport %s -j TTL --ttl-set %s' % (port_list[socket_number], ttl_socket[socket_number]))
    os.system('iptables -t mangle -A POSTROUTING -p TCP --tcp-flags FIN FIN --
dport %s -j TTL --ttl-set %s' % (port_list[socket_number], ttl_socket[socket_number]))

    print("Now ttl of socket ", socket_number, "is: ", ttl_socket[socket_number])

def send_data(data_socket, port, socket_number, data_begin, data_end):

    global data
    global ttl_list
    global ttl_socket
    global first
    global total_sent
    global all_sockets_total_sent

    while True:
        try:
            data_socket.connect((receiver_IP, port))
            break
        except socket.error:
            pass

    data_socket.settimeout(1)
    start_socket_time = time.time()

    while total_sent[socket_number] < data_end - data_begin:

        try:
            # SEND
            sent = data_socket.send(data[data_begin + total_sent[socket_number]:
data_end])

            all_sockets_total_sent += sent

            current_time = time.time() - start_time #Time from the beginning of the
transmission
            current_socekt_time = time.time() - start_socket_time
            log.write('Socket ' + str(socket_number) + "\t" + str(round(current_time,2)) + '
seconds\t' + str(round(all_sockets_total_sent,2)) + ' Bytes\n')
            total_sent[socket_number] += sent

        except socket.timeout:
            os.system('iptables -t mangle -D POSTROUTING -p TCP --tcp-flags ACK
ACK --dport %s -j TTL --ttl-set %s' % (port_list[socket_number],
ttl_socket[socket_number]))
            os.system('iptables -t mangle -D POSTROUTING -p TCP --tcp-flags FIN FIN
--dport %s -j TTL --ttl-set %s' % (port_list[socket_number],
ttl_socket[socket_number]))

            ttl_socket[socket_number] = None
            change_ttl(socket_number)
            time.sleep(0.1)

    socket_time[socket_number] = time.time() - start_socket_time
    sent_per_socket[socket_number] = 'Sent per socket ' + str(socket_number) + ': ' +
str(total_sent[socket_number]) + '. In: ' + str(round(socket_time[socket_number],2)) +
' seconds'
    print('Sent per socket ' + str(socket_number) + ': ' + str(total_sent[socket_number])
+ '. In: ' + str(round(socket_time[socket_number],2)) + ' seconds')

    data_socket.close()
    print("Close socket ", socket_number)

    total_sent_copy = total_sent

    total_sent_aux = total_sent_copy[0]
    min_thoughput_index = 0

    for i in range (1, NUMBER_OF_SOCKETS):

        if total_sent_copy[i] < total_sent_aux:
            total_sent_aux = total_sent_copy[i]
            min_thoughput_index = i

    os.system('iptables -t mangle -D POSTROUTING -p TCP --tcp-flags ACK ACK --
dport %s -j TTL --ttl-set %s' % (port_list[min_thoughput_index],
ttl_socket[min_thoughput_index]))
    os.system('iptables -t mangle -D POSTROUTING -p TCP --tcp-flags FIN FIN --
dport %s -j TTL --ttl-set %s' % (port_list[min_thoughput_index],
ttl_socket[min_thoughput_index]))

    ttl_socket[min_thoughput_index] = ttl_socket[socket_number]

    os.system('iptables -t mangle -A POSTROUTING -p TCP --tcp-flags ACK ACK --
dport %s -j TTL --ttl-set %s' % (port_list[min_thoughput_index],
ttl_socket[min_thoughput_index]))
    os.system('iptables -t mangle -A POSTROUTING -p TCP --tcp-flags FIN FIN --
dport %s -j TTL --ttl-set %s' % (port_list[min_thoughput_index],
ttl_socket[min_thoughput_index]))

    print("changed ttl of socket ", min_thoughput_index)
```

```
if __name__ == '__main__':

   # Process command line args
   try:
      filename = sys.argv[1]
      receiver_IP = socket.gethostbyname(sys.argv[2])
      log_file_name = sys.argv[3]

   except IndexError as TypeError:
      exit('Usage: ./sender.py <filename> <receiver_IP> <log_file_name>')

   os.system('iptables -t mangle -F POSTROUTING') # Remove all possible
POSTROUTING rules

   socket_list = {} # List to store the sockets
   port_list = {}
   ttl_list = {}

   ttl_socket = {}

   socket_time = {}

   ttl_list[0] = 127
   ttl_list[1] = 160
   ttl_list[2] = 131
   ttl_list[3] = 130
   ttl_list[4] = 161
   ttl_list[5] = 128
   ttl_list[6] = 127
   ttl_list[7] = 160
   ttl_list[8] = 131
   ttl_list[9] = 161

   for i in range(0, NUMBER_OF_SOCKETS):
      ttl_socket[i] = ttl_list[i]

   port = 12345 # First port that is used

   # Open file to read its bytes
   try:
      file_in = open(filename, 'rb')
   except:
      exit("Could not open " + filename)

   # Store the bytes of the file
   data = file_in.read()
   file_in.close()

   data_length = len(data)
   data_per_socket = math.ceil(data_length / NUMBER_OF_SOCKETS)
```

```
  # Open file to store logs
  log = open(log_file_name, 'w')

  first = False
  processes = []
  total_sent = [0] * NUMBER_OF_SOCKETS
  all_sockets_total_sent = 0
  sent_per_socket = {}
  start_time = time.time()

  # Create all the sockets
  for i in range (0, NUMBER_OF_SOCKETS):

      data_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Create
TCP socket
      data_socket.setblocking(False)

      socket_list[i] = data_socket # Store in the socker buffer the socket with its
corresponding port and ttl

      port_list[i] = port + i

      os.system('iptables -t mangle -A POSTROUTING -p TCP --tcp-flags ACK ACK
--dport %s -j TTL --ttl-set %s' % (port_list[i], ttl_socket[i]))
      os.system('iptables -t mangle -A POSTROUTING -p TCP --tcp-flags FIN FIN --
dport %s -j TTL --ttl-set %s' % (port_list[i], ttl_socket[i]))

      print('Socket ', i, ' connected to IP ', receiver_IP, ' through port ', port_list[i])

      data_begin = int(data_per_socket * i)

      if i == NUMBER_OF_SOCKETS - 1:
         data_end = data_length
      else:
         data_end = int(data_per_socket * i + data_per_socket)

      print("Data to be sent through socket ", i, ": ", data_end - data_begin)
      processes.append(Thread(target = send_data, args = (socket_list[i], port_list[i], i,
data_begin, data_end)))

  print ('\nCreated all sockets correctly\n')

  for proc in processes:
      proc.start()

  print("\nSTART SENDING DATA\n")

  for proc in processes:
      proc.join()
```

```
    total_time = time.time() - start_time

    log.write('\n')

    for i in range(0, NUMBER_OF_SOCKETS):
        log.write(sent_per_socket[i])
        socket_list[i].close()
        print("Close socket ", i)

    print("\nTOTAL TIME:", round(total_time,2))

    log.write('\n----------- END of Transmission -----------\n')
    log.write('Total time: ' + str(round(total_time,2)) + '\n')
    log.close()

    for i in range(0, NUMBER_OF_SOCKETS):
        os.system('iptables -t mangle -D POSTROUTING -p TCP --tcp-flags ACK ACK
--dport %s -j TTL --ttl-set %s' % (port_list[i], ttl_socket[i]))
        os.system('iptables -t mangle -D POSTROUTING -p TCP --tcp-flags FIN FIN --
dport %s -j TTL --ttl-set %s' % (port_list[i], ttl_socket[i]))

    sys.exit("----------- END of Transmission -----------")
```

Appendix B. Multi-socket-TCP receiver code

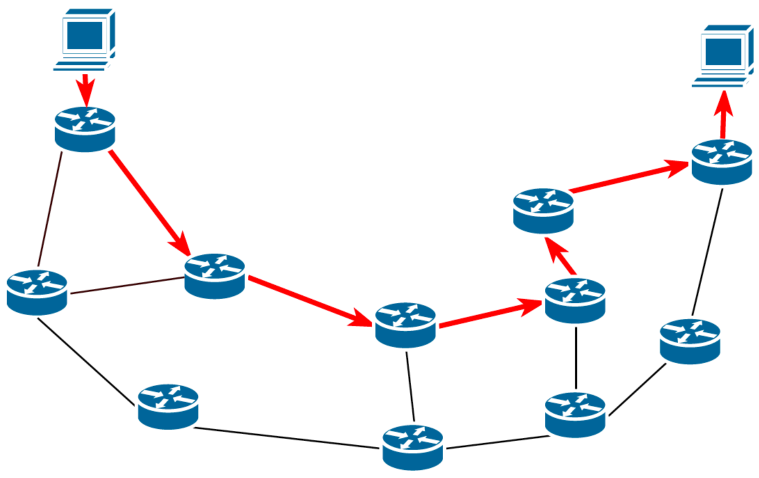
```
# receiver_multi_tcp_threads.py

import socket
import time
import sys
import math
import os
from threading import Thread

PAYLOAD_SIZE = 4096
NUMBER_OF_SOCKETS = 4

def change_ttl(socket_number):

   global destination_port_list
   global ttl_list
   global ttl_socket

   count = 0

   i = socket_number + 1

   if i == NUMBER_OF_SOCKETS:
      i = 0

   ttl_temp = ttl_socket[socket_number]

   while count < NUMBER_OF_SOCKETS - 1:

      if ttl_socket[i] is not None:
         ttl_socket[socket_number] = ttl_socket[i]
         break

      i = i + 1

      if i == NUMBER_OF_SOCKETS:
         i = 0
```

```
        count = count + 1

    if ttl_socket[socket_number] == None or ttl_temp == ttl_socket[socket_number]:
        for i in range (0, NUMBER_OF_SOCKETS):
            ttl_socket[i] = ttl_list[i]

    os.system('iptables -t mangle -A POSTROUTING -p TCP --tcp-flags ACK ACK --
dport %s -j TTL --ttl-set %s' % (destination_port_list[socket_number],
ttl_socket[socket_number]))
    os.system('iptables -t mangle -A POSTROUTING -p TCP --tcp-flags FIN FIN --
dport %s -j TTL --ttl-set %s' % (destination_port_list[socket_number],
ttl_socket[socket_number]))

    print("Now ttl of socket ", socket_number, "is: ", ttl_socket[socket_number])

def check_to_write():

    global received_file
    global received_file_part

    for i in range (0, NUMBER_OF_SOCKETS):

        file = open('temp' + str(i), 'rb')
        data = file.read()
        file.close()

        received_file.write(data)

        received_file_part[i].close()
        os.remove("temp" + str(i))

        print("Stored data received from socket ", i, ". Size: ", len(data))

def receive_data(data_socket, socket_number):

    global destination_port_list
    global ttl_socket
    global first
    global total_received

    while True:
        try:
            conn, addr = data_socket.accept()     # Establish connection with client.
            break
        except socket.error:
            pass

    print('Socket ', socket_number, ' connected to IP ', addr[0], ' through port ', addr[1])
    socket_connection_list[socket_number] = conn
    destination_port_list[socket_number] = addr[1]

    os.system('iptables -t mangle -A POSTROUTING -p TCP --tcp-flags ACK ACK --
dport %s -j TTL --ttl-set %s' % (destination_port_list[socket_number],
ttl_socket[socket_number]))
    os.system('iptables -t mangle -A POSTROUTING -p TCP --tcp-flags FIN FIN --
dport %s -j TTL --ttl-set %s' % (destination_port_list[socket_number],
ttl_socket[socket_number]))

    timeout_set = False

    while True:

        try:
            data = conn.recv(PAYLOAD_SIZE)

            if not timeout_set:
                timeout_set = True
                conn.settimeout(2)

            if not data:
                break

            received_file_part[socket_number].write(data)

            total_received[socket_number] += len(data)

        except socket.timeout:
            os.system('iptables -t mangle -D POSTROUTING -p TCP --tcp-flags ACK
ACK --dport %s -j TTL --ttl-set %s' % (destination_port_list[socket_number],
ttl_socket[socket_number]))
            os.system('iptables -t mangle -D POSTROUTING -p TCP --tcp-flags FIN FIN
--dport %s -j TTL --ttl-set %s' % (destination_port_list[socket_number],
ttl_socket[socket_number]))

            ttl_socket[socket_number] = None

            change_ttl(socket_number)

            time.sleep(0.1)

    received_file_part[socket_number].close()
    socket_list[socket_number].close()
    print("Close socket ", socket_number)

    total_received_copy = total_received

    total_received_aux = total_received_copy[0]
    min_thoughput_index = 0
    for i in range (1, NUMBER_OF_SOCKETS):

        if total_received_copy[i] < total_received_aux:
            total_received_aux = total_received_copy[i]
            min_thoughput_index = i

    os.system('iptables -t mangle -D POSTROUTING -p TCP --tcp-flags ACK ACK --
dport %s -j TTL --ttl-set %s' % (destination_port_list[min_thoughput_index],
ttl_socket[min_thoughput_index]))
    os.system('iptables -t mangle -D POSTROUTING -p TCP --tcp-flags FIN FIN --
dport %s -j TTL --ttl-set %s' % (destination_port_list[min_thoughput_index],
ttl_socket[min_thoughput_index]))

    ttl_socket[min_thoughput_index] = ttl_socket[socket_number]

    os.system('iptables -t mangle -A POSTROUTING -p TCP --tcp-flags ACK ACK --
dport %s -j TTL --ttl-set %s' % (destination_port_list[min_thoughput_index],
ttl_socket[min_thoughput_index]))
    os.system('iptables -t mangle -A POSTROUTING -p TCP --tcp-flags FIN FIN --
dport %s -j TTL --ttl-set %s' % (destination_port_list[min_thoughput_index],
ttl_socket[min_thoughput_index]))

    print("changed ttl of socket ", min_thoughput_index)

if __name__ == '__main__':

    # Process command line args
    try:
        filename = sys.argv[1]
        receiver_IP = socket.gethostbyname(sys.argv[2])

    except IndexError as TypeError:
        exit('Usage: ./receiver.py <filename> <receiver_IP>')

    os.system('iptables -t mangle -F POSTROUTING') # Remove all possible
POSTROUTING rules

    socket_list = {} # List to store the sockets
    socket_connection_list = {}
    port_list = {}
    destination_port_list = {}
    ttl_list = {}

    ttl_socket = {}

    ttl_list[0] = 127
    ttl_list[1] = 160
    ttl_list[2] = 131
    ttl_list[3] = 176
    ttl_list[4] = 144
    ttl_list[5] = 146
    ttl_list[6] = 132
    ttl_list[7] = 176
    ttl_list[8] = 127
    ttl_list[9] = 160

    for i in range(0, NUMBER_OF_SOCKETS):
        ttl_socket[i] = ttl_list[i]

    port_base = 12345 # First port that is used

    # Open file to write the file
    received_file = open(filename, 'wb')

    received_file_part = {}
    file_seqnum = 0

    total_received = [0] * NUMBER_OF_SOCKETS
    processes = []

    print ('Waiting for the sender\n')

    for i in range (0, NUMBER_OF_SOCKETS):

        data_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Create
TCP socket
        data_socket.setblocking(False)

        socket_list[i] = data_socket # Still don't have connection (2 missing values at this
point, but it doesn't matter)

        port_list[i] = port_base + i

        socket_list[i].bind((receiver_IP, port_list[i])) # Bind to the port
        socket_list[i].listen(5) # Now wait for client connection.

        received_file_part[i] = open("temp" + str(i), 'wb')
        processes.append(Thread(target = receive_data, args = (socket_list[i], i)))

    for proc in processes:
        try:
            proc.start()
        except KeyboardInterrupt:
            print("caught keyboard interrupt, exiting")

    for proc in processes:
        proc.join()

    check_to_write()
    received_file.close()

    for i in range(0, NUMBER_OF_SOCKETS):
        socket_connection_list[i].close()
        print("Close socket ", i)

    for i in range(0, NUMBER_OF_SOCKETS):
        os.system('iptables -t mangle -D POSTROUTING -p TCP --tcp-flags ACK ACK
--dport %s -j TTL --ttl-set %s' % (destination_port_list[i], ttl_socket[i]))
        os.system('iptables -t mangle -D POSTROUTING -p TCP --tcp-flags FIN FIN --
dport %s -j TTL --ttl-set %s' % (destination_port_list[i], ttl_socket[i]))
    #os.system('iptables -t mangle -F POSTROUTING') # Remove all possible
POSTROUTING rules

    sys.exit("----------- END of Transmission -----------")
```

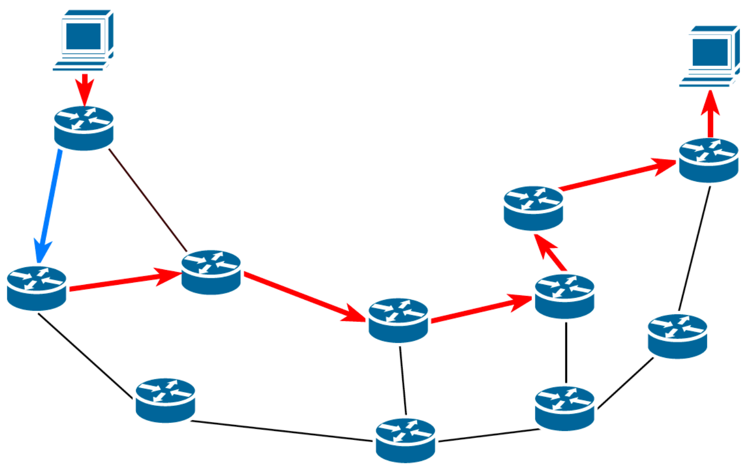
Appendix C. Paths from "ubuntu-1" to "ubuntu-2" in Abilene topology

Links in red colour represent the primary next hop, whereas links in blue represent the secondary next hop.

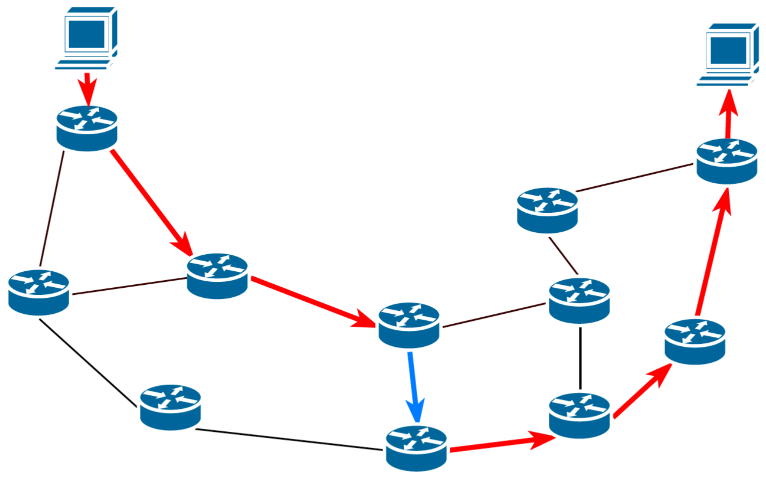
Shortest path. TTL ∊ [6, 127], [133, 143], [149, 159], [165, 175], [181, 191], [197, 207], [213, 223], [229, 239], [245, 255]



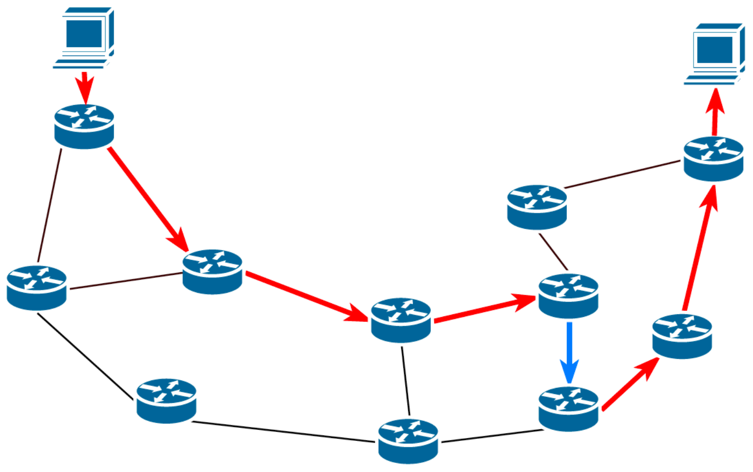
TTL = 128



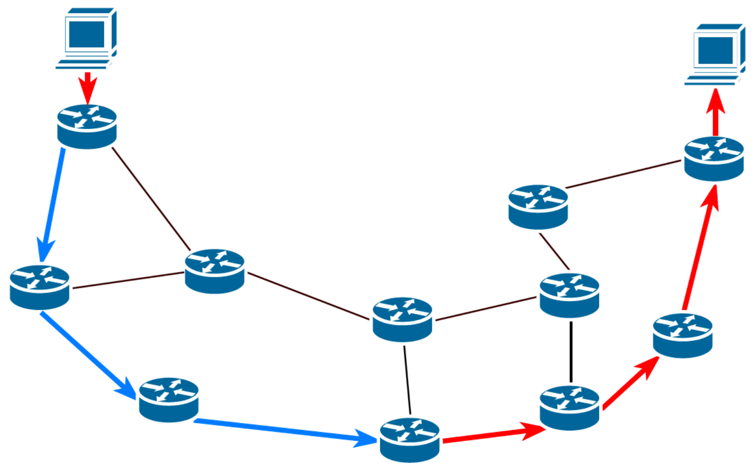
TTL = 130



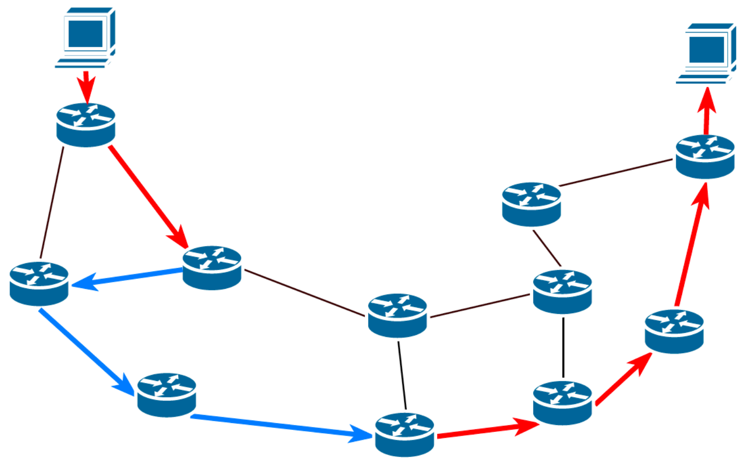
TTL = 131



TTL = 160, 176, 192, 208, 224, 240



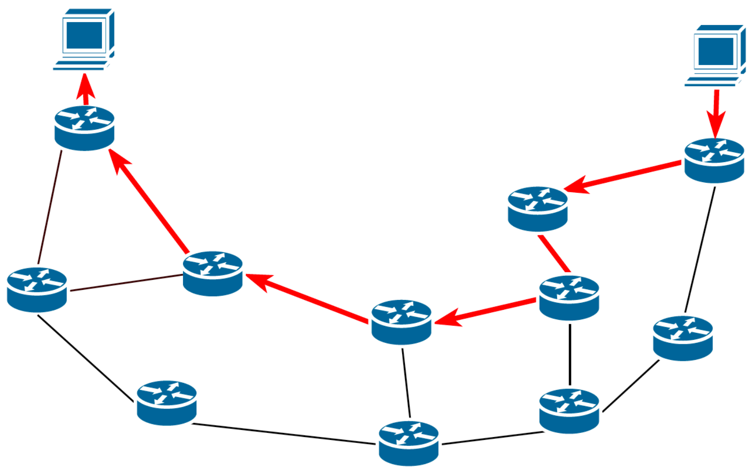
TTL = 161, 177, 193, 209, 225, 241



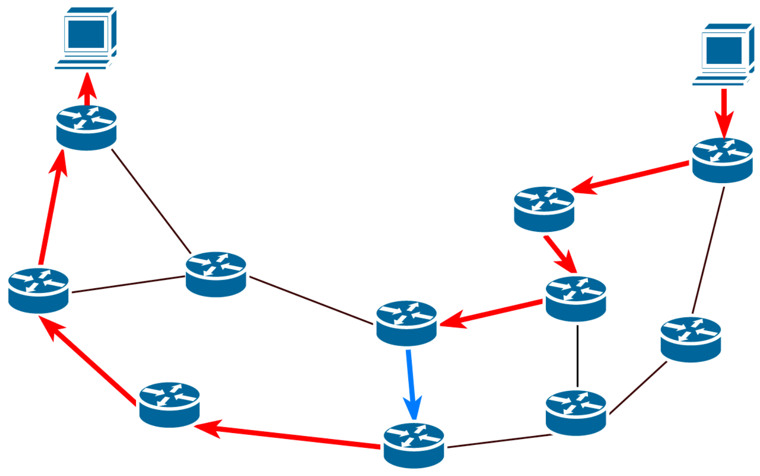
Appendix D. Paths from "ubuntu-2" to "ubuntu-1" in Abilene topology

Links in red colour represent the primary next hot, whereas links in blue represent the secondary next hop.

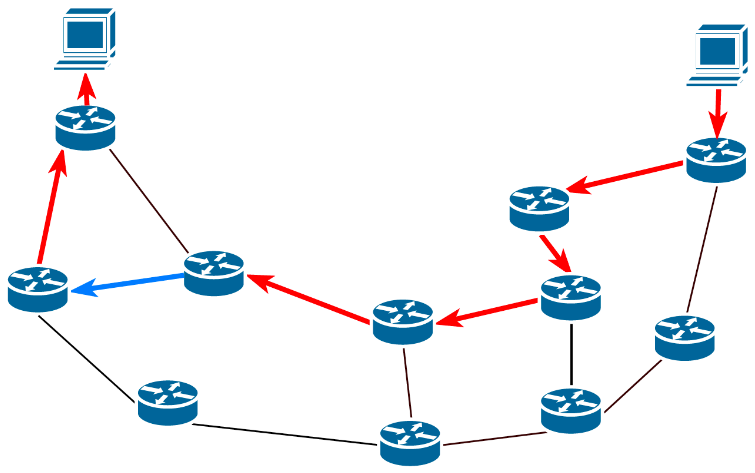
Shortest path. TTL ∊ [6, 127], [133, 143], [149, 159], [165, 175], [181, 191], [197, 207], [213, 223], [229, 239], [245, 255]



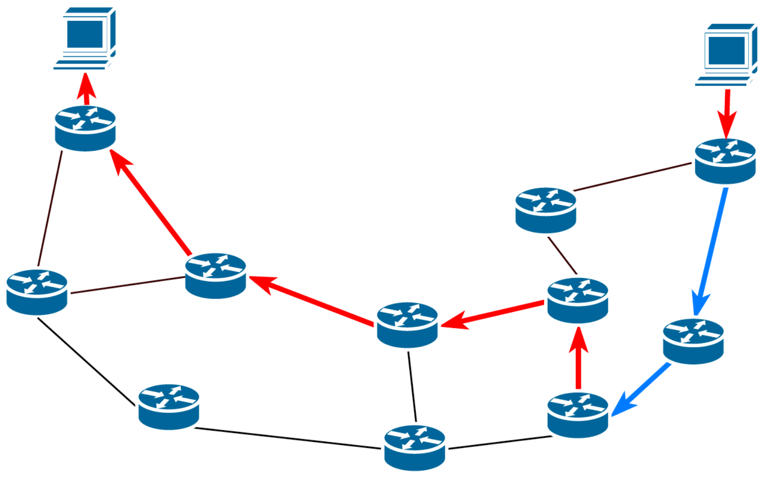
TTL = 131



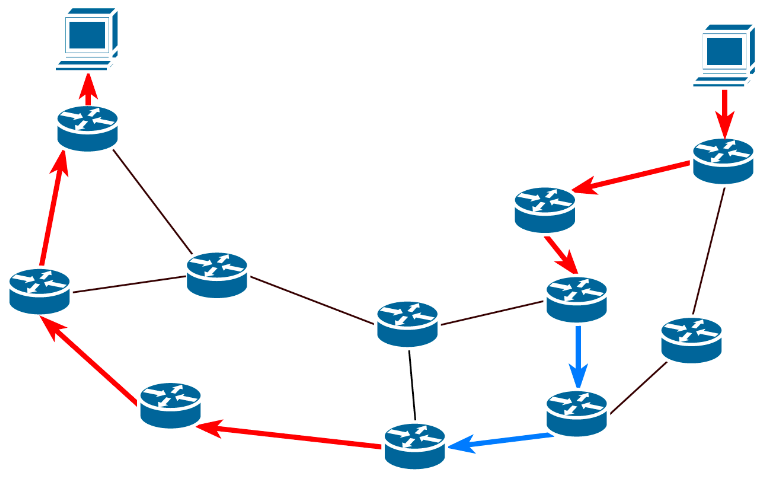
TTL = 132, 148, 164, 180, 196, 212, 228, 244



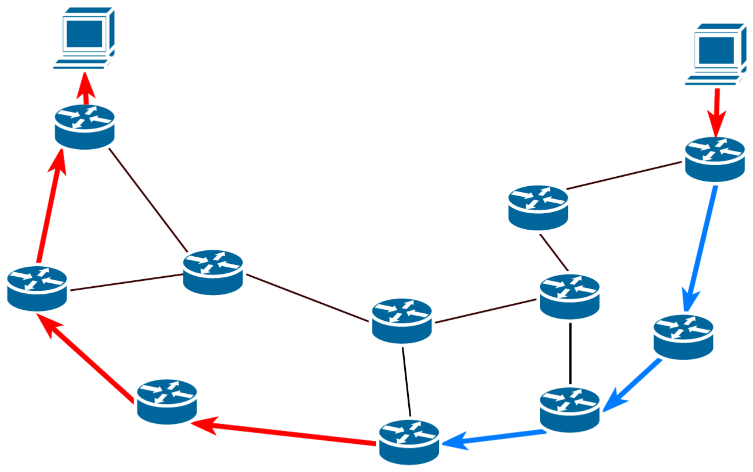
TTL = 144



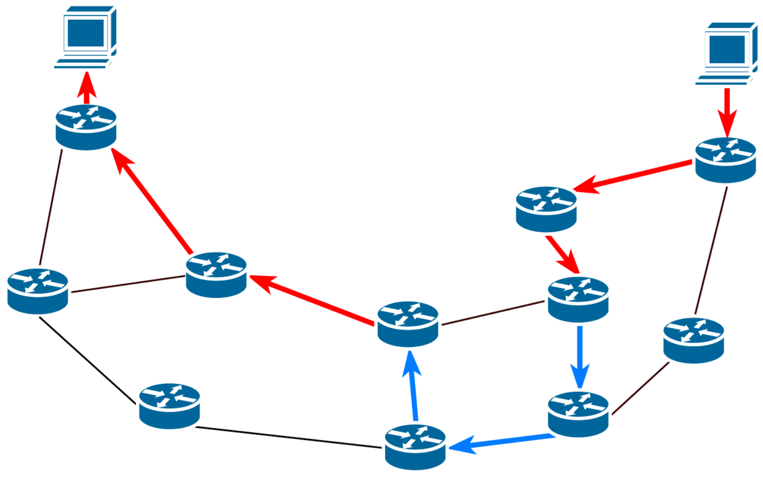
TTL = 146



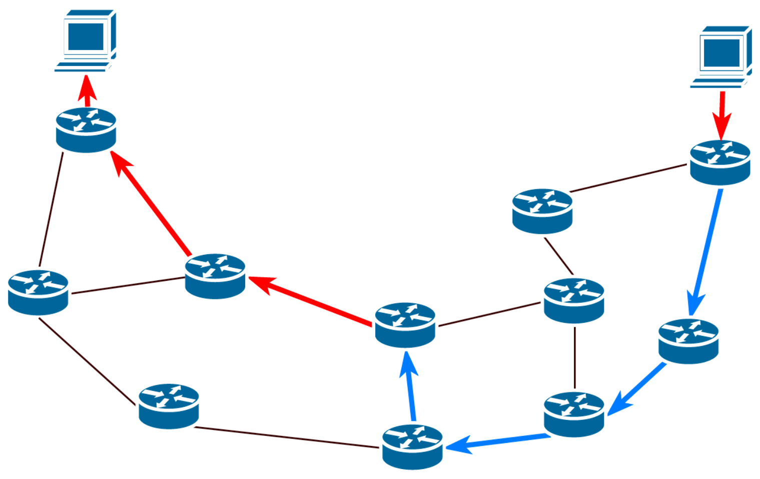
TTL = 160



TTL = 162, 178, 194, 210, 226, 242



TTL = 176, 192, 208, 224, 240



List of Figures

Figure 2.1. TCP connection ... 11
Figure 2.2. iptables routing and packet filtering process [12] ... 14
Figure 2.3. FRR architecture [14] ... 16
Figure 2.4. OSPF API architecture [15] ... 17
Figure 3.1. Global view of router behaviour [5] ... 19
Figure 3.2. Simple topology ... 23
Figure 3.3. “frr-ubuntu-1” routing tables ... 27
Figure 3.4. "ubuntu-1" console with the ping through the primary path ... 27
Figure 3.5. Wireshark showing the ping through the primary path between routers “frr-router-1” and “frr-router-2” ... 28
Figure 3.6. "ubuntu-1" console with the unsuccessful ping through the secondary path ... 28
Figure 3.7. Wireshark showing the unsuccessful ping through the secondary path between routers “frr-router-1” and “frr-router-3” ... 29
Figure 3.8. Wireshark showing the unsuccessful ping through the secondary path between routers “frr-router-3” and “frr-router-2” ... 29
Figure 3.9. Wireshark showing the unsuccessful ping through the secondary path between router “frr-router-2” and host “ubuntu-2” ... 30
Figure 3.10. "ubuntu-1" console with the successful ping through the secondary path ... 30
Figure 3.11. Wireshark showing the successful ping through the secondary path between routers “frr-router-1” and “frr-router-3” (a); and between routers “frr-ubuntu-1” and frr-ubuntu-2 (b) ... 31
Figure 3.12. Multi-TCP-sockets sender flow diagram ... 34
Figure 3.13. Multi-TCP-sockets receiver flow diagram ... 37
Figure 3.14. Abilene topology [1] ... 39
Figure 3.15. Abilene topology in GNS3 ... 41
Figure 4.1. Use Case 1. Comparison of the bigger file transmitted using multipath and single path ... 43
Figure 4.2. Use Case 1. Comparison of the smaller file transmitted using multipath and single path ... 44
Figure 4.3. Overloaded links in Use Case 2 ... 45
Figure 4.4. Use Case 2. Comparison of the bigger file transmitted using multipath and single path ... 46
Figure 4.5. Use Case 2. Comparison of the smaller file transmitted using multipath and single path ... 47
Figure 4.6. Link to be removed in Use Case 3 ... 48
Figure 4.7. Use Case 3. Comparison of the file bigger transmitted using multipath and single path ... 49

Figure 4.8. Use Case 3. Comparison of the smaller file transmitted using multipath and single path..50
Figure 4.9. Comparison of multipath in the 3 use cases when transmitting the bigger file..51
Figure 4.10. Comparison of multipath in the 3 use cases when transmitting the smaller file..51
Figure 4.11. Secondary – Primary loop 54
Figure 4.12. Secondary – Secondary loop 54
Figure 4.13. Primary – Secondary – Primary loop 54

List of Tables

Table 3.1. Forwarding rules 24
Table 3.2. Matching TTL values 24
Table 4.1. Transmitted files size 43
Table 4.2. Use Case 1. Bigger file transmission time 43
Table 4.3. Use Case 1. Smaller file transmission time 44
Table 4.4. Use Case 2. Bigger file transmission time 46
Table 4.5. Use Case 2. Smaller file transmission time 46
Table 4.6. Use Case 3. Bigger file transmission time 48
Table 4.7. Use Case 3. Smaller file transmission time 49
Table 4.8. Types of loops from host “ubuntu-1” to host “ubuntu-2” 53
Table 4.9. Types of loops from host “ubuntu-2” to host “ubuntu-1” 55

Abbreviations

CIDR	Classless-Interdomain Routing
GNS3	Graphical Network Simulator 3
GUI	Graphical User Interface
FRR	Free Range Routing
IMRE	Ideal Multipath Routing Expedient
ISP	Internet Service Provider
LSA	Link State Advertisement
LSDB	Link State Data Base
MAC	Mandatory Access Control
MD5	Message Digest 5
OS	Operating System
OSPF	Open Shortest Path First
PID	Process IDentification number
TCP	Transport Control Protocol
TTL	Time To Live
VLSM	Variable Length Subnet Mask
VM	Virtual Machine

Bibliography

[1] Xiaowei Yang and David Wetherall. *Source Selectable Path Diversity via Routing Deflections*. In Proc. *ACM* SIGCOMM, pp. 159-170, 2006, Pisa, Italy.

[2] Matthew Caesar, Martín Casado, Teemu Koponen, Jennifer Rexford, Scott Shenker. *Dynamic Route Computation Considered Harmful*. ACM SIGCOMM Computer Communication Review, Volume 40, Number 2, pp. 66-71, April 2010.

[3] Murtaza Motiwala, Megan Elmore, Nick Feamster and Santosh Vempala. *Path Splicing*. SIGCOMM'08, August 17–22, pp. 26-38, 2008, Seattle, Washington, USA.

[4] Christoph Paasch. *Improving Multipath TCP*. PhD Thesis, October 30, 2014, Belgium.

[5] Lajos Zongor. *Building a measurement environment for multi-path routing (in Hungarian)*, Project report, Budapest University of Technology and Economics, 2018.

[6] A. Markopoulou, G. Iannaconne, S. Bhattacharrya, C.-N. Chuah, Y. Ganjali and C. Diot. *Characterization of failures in an operational IP backbone network*. In IEEE/ACM Trans. Networking, vol. 16, no 4, pp 749-762, August 2008.

[7] Sébastien Barré, Christoph Paasch, Olivier Bonaventure. *MultiPath TCP: From Theory to Practice*. 10th IFIP Networking Conference (NETWORKING), May 2011, Valencia, Spain. Springer, Lecture Notes in Computer Science, LNCS-6640 (Part I), pp.444-457, 2011, NETWORKING 2011.

[8] Bert Hubert, Gregory Maxwell, Remco van Mook, Martijn van Oosterhout, Paul B Schroeder, Jasper Spaans. *Linux Advanced Routing & Traffic Control HOWTO*. Revision 1.1, 2002−07−22.

[9] The official guide and reference for GNS3, *https://docs.gns3.com*, [Online], Accessed: 2019.04.20.

[10] Getting Started with GN3, *https://docs.gns3.com/1PvtRW5eAb8RJZ11maEYD9_aLY8kkdhgaMB0wPCz8a38/index.html*, [Online], Accessed: 2019.04.20.

[11] Docker documentation, *https://www.docs.docker.com*, [Online], Accessed: 2019.04.20.

[12] *https://wiki.archlinux.org/index.php/Iptables*, [Online], Accessed: 2019.04.20.

[13] Iptables manual page, *https://jlk.fjfi.cvut.cz/arch/manpages/man/iptables.8*, [Online], Accessed: 2019.04.20.

[14] FRRouting User Guide*,* *http://docs.frrouting.org*, [Online], Accessed: 2019.04.20.

[15] OSPF API Documentation, *http://docs.frrouting.org/projects/dev-guide/en/latest/ospf-api.html*, [Online], Accessed: 2019.04.20.

[16] LEMON: A C++ Library for Efficient Modeling and Optimization in Networks, *http://lemon.cs.elte.hu*. [Online], Accessed: 2019.04.20.