Evaluation of audiovisual streaming algorithms using the FEC technique over VNX virtual networks

A Master’s Thesis

Submitted in partial fulfilment of the requirements for the Master’s Degree in Telecommunications Engineering

by

Lorenzo Pantaleo

Advisor: Prof. Jorge Mata Diaz,

Barcelona, July 2018
Title of the thesis
Evaluation of audiovisual streaming algorithms using the FEC technique with VNX virtual networks

Author
Lorenzo Pantaleo

Advisor
Prof. Jorge Mata Diaz

Date
Barcelona, July 2018

This thesis was developed for
Escola Tcnica d’Enginyeria de Telecomunicaci de Barcelona, Universitat Politcnica de Catalunya

in collaboration with the
Departament dEnginyeria Telematica
Abstract

The developed project consists in implementing and testing the performances of some of the algorithms that can be adopted for the streaming of audiovisual contents in satellite networks. The implementation of the evaluated system has been developed in virtual networks using the open-source virtualization tool (VNX). This tool allows several virtual machines running Linux operating system to be started and virtually connected to each other. The algorithms that have been evaluated in this work are intended to be used to enhance the responsivity performances of audiovisual streaming applications over high-latency lossy environments (e.g. satellite networks), without affecting the reliability of the system. More specifically, the aim of those algorithms is to implement various kinds of FEC techniques and congestion control mechanisms, instead of the classical TCP mechanisms, which may significantly affect the overall performances as a consequence of the just described features of the environment itself.

Keywords
Virtual networks, Linux, Performance Enhancing Proxy algorithms, Video Broadcasting, Satellite Networks
# Contents

Abstract i

1 Introduction 1
   1.1 Objectives ......................................................... 2
       1.1.1 Requirements and Specifications .......................... 3
   1.2 Methods and Procedures review ................................. 3
       1.2.1 Third-party resources ....................................... 4
   1.3 Thesis outline .................................................... 5
       1.3.1 Workflow, methodology and tools ............................ 5

2 Theoretical background 7
   2.1 Satellite communications overview ............................. 7
       2.1.1 Modeling the scenario ....................................... 8
   2.2 Transport-layer protocols recalls ............................... 9
       2.2.1 The User Datagram Protocol ................................ 9
       2.2.2 The Transmission Control Protocol ......................... 11
   2.3 Transport-layer issues in satellite networks .................. 21
       2.3.1 The Mathis equation ......................................... 21
       2.3.2 TCP improved schemes for wireless lossy networks ....... 25
   2.4 FEC techniques ..................................................... 28
       2.4.1 Network Coding ............................................... 29
       2.4.2 Reed-Solomon ................................................ 32

3 State of the art of the used technologies 35
   3.1 Used software ....................................................... 35
       3.1.1 LXC .......................................................... 35
       3.1.2 VNX .......................................................... 38
       3.1.3 Iperf ......................................................... 48
CONTENTS

3.1.4 NGINX .................................................. 49
3.2 Studied Algorithms ....................................... 51
  3.2.1 UDPspeeder and tinyfecVPN .......................... 51
  3.2.2 PEPsal ................................................ 54
  3.2.3 TCPeP ............................................... 56

4 Experimental setup ........................................ 63
  4.1 Building the scenario .................................... 65
  4.2 Performance evaluations ................................ 67
    4.2.1 Latency test ....................................... 67
    4.2.2 Throughput test .................................... 71
  4.3 Results Comparison .................................... 105
    4.3.1 Numerical results .................................. 105
    4.3.2 Video streaming comparison ......................... 110

5 Conclusions and future developments ....................... 115
  5.1 Conclusions ............................................. 115
  5.2 Future developments .................................... 116

Appendices .................................................. 119

A xml codes used to generate the VNX scenarios .............. 121
  A.1 Basic two-hosts scenario ............................... 121
  A.2 TCPeP trial scenario ................................... 122
  A.3 Complete scenario ...................................... 124

B List of commands used to realize the performed test ...... 133
  B.1 Setting the lossy link and the channel capacity ....... 133
  B.2 Iperf file transfers .................................... 133

C The TCPeP congestion control algorithm .................... 135
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The UDP header</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>The TCP header</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Representation of the Sliding-Window control mechanism</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Sequence diagram of a TCP Client-receiver message exchange, using a sliding window of 3 packets</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>The TCP flow control mechanism</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>Sequence diagram of a TCP sender-receiver message exchange in case of 3 DUPACK reception</td>
<td>16</td>
</tr>
<tr>
<td>2.7</td>
<td>Sequence diagram of a TCP sender-receiver message exchange in case of RTO expiration</td>
<td>16</td>
</tr>
<tr>
<td>2.8</td>
<td>cwnd waveform during the various congestion control phases</td>
<td>18</td>
</tr>
<tr>
<td>2.9</td>
<td>cwnd waveform using TCP Reno assuming 3 DUPACK losses only</td>
<td>23</td>
</tr>
<tr>
<td>2.10</td>
<td>cwnd waveform using TCP Reno</td>
<td>25</td>
</tr>
<tr>
<td>2.11</td>
<td>cwnd waveform using TCP New Reno</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>Internal LXC LAN configuration</td>
<td>36</td>
</tr>
<tr>
<td>3.2</td>
<td>Summarization of the structure of a VNX configuration file</td>
<td>43</td>
</tr>
<tr>
<td>3.3</td>
<td>Scenario created by the vnx software using scenario1.xml</td>
<td>44</td>
</tr>
<tr>
<td>3.4</td>
<td>Scheme of the internal LAN organization of this scenario</td>
<td>47</td>
</tr>
<tr>
<td>3.5</td>
<td>netcat message exchange through the vbridge network</td>
<td>48</td>
</tr>
<tr>
<td>3.6</td>
<td>Schematic representation of the behaviour of the UDPspeeder algorithm</td>
<td>52</td>
</tr>
<tr>
<td>3.7</td>
<td>Schematic representation of the behaviour of the UDPspeeder algorithm over an UDP tunnel</td>
<td>53</td>
</tr>
<tr>
<td>3.8</td>
<td>Schematization of the FEC code appending and UDP tunnelling processes</td>
<td>54</td>
</tr>
<tr>
<td>3.9</td>
<td>Sequence diagram summarizing the PEPsal algorithm behaviour</td>
<td>57</td>
</tr>
<tr>
<td>3.10</td>
<td>Scenario map created using scenario2.xml</td>
<td>58</td>
</tr>
<tr>
<td>3.11</td>
<td>Packets exchanged through the loopback interface</td>
<td>60</td>
</tr>
</tbody>
</table>
3.12 Packets exchanged through the feeder interface ........................................ 61
3.13 Packets exchanged through the sinker interface ........................................ 61

4.1 The overall scenario in a splitted satellite network ........................................ 65
4.2 Tested scenario ............................................................................................... 66
4.3 Schematic representation of the kernel-level architecture ............................... 66
4.4 Flow graph of Token Bucket algorithm ......................................................... 69
4.5 Traffic control settings in saterm and hubm .................................................... 70
4.6 Packets captured by the feeder device ......................................................... 73
4.7 Packets captured by the satellite device ....................................................... 73
4.8 Packets captured by the remotebr device ..................................................... 73
4.9 Sequence diagram summarization of the previous captures ............................ 74
4.10 Stevens diagrams obtained by performing a short transfer in the Uplink direction with a packet error probability amount of $10^{-2}, 10^{-3}$ and $10^{-4}$ .................. 75
4.11 Throughput curves obtained by performing the medium transfer in the Uplink direction with a packet error probability amount of respectively $10^{-2}, 10^{-3}$ and $10^{-4}$ 77
4.12 Throughput curves obtained by performing the long transfer in the Uplink direction with a packet error probability amount of respectively $10^{-2}, 10^{-3}$ and $10^{-4}$ .... 78
4.13 Stevens diagrams obtained by performing a short transfer in the Downlink direction with a packet error probability amount of $10^{-2}, 10^{-3}$ and $10^{-4}$ .......................... 79
4.14 Throughput curves obtained by performing the medium transfer in the Downlink direction with a packet error probability amount of respectively $10^{-2}, 10^{-3}$ and $10^{-4}$ 81
4.15 Throughput curves obtained by performing the long transfer in the Downlink direction with a packet error probability amount of respectively $10^{-2}, 10^{-3}$ and $10^{-4}$ .... 82
4.16 Stevens diagrams obtained by performing a short transfer in the Uplink direction with a packet error probability amount of $10^{-2}, 10^{-3}$ and $10^{-4}$ .......................... 85
4.17 Throughput curves obtained by performing the medium transfer in the Uplink direction with a packet error probability amount of respectively $10^{-2}, 10^{-3}$ and $10^{-4}$ 87
4.18 Throughput curves obtained by performing the long transfer in the Uplink direction with a packet error probability amount of respectively $10^{-2}, 10^{-3}$ and $10^{-4}$ .... 88
4.19 Throughput curves obtained by performing the medium transfer in the Downlink direction with a packet error probability amount of respectively $10^{-2}, 10^{-3}$ and $10^{-4}$ 89
4.20 Throughput curves obtained by performing the long transfer in the Downlink direction with a packet error probability amount of respectively $10^{-2}, 10^{-3}$ and $10^{-4}$ .... 90
4.21 Packets captured by the remotebr device ................................................... 94
4.22 Packets captured by the remotebr device ................................................... 94
LIST OF FIGURES

4.23 Packets captured by the remotebr device ........................................... 94
4.24 Sequence diagram summarization of the previous captures ....................... 95
4.25 Stevens diagrams obtained by performing a short transfer in the Uplink direction
   with a packet error probability amount of $10^{-2}$, $10^{-3}$ and $10^{-4}$ .................. 97
4.26 Throughput curves obtained by performing the medium transfer in the Uplink
direction with a packet error probability amount of respectively $10^{-2}$, $10^{-3}$ and $10^{-4}$ 98
4.27 Throughput curves obtained by performing the long transfer in the Uplink
direction with a packet error probability amount of respectively $10^{-2}$, $10^{-3}$ and $10^{-4}$ . . . . 99
4.28 Throughput curves obtained by performing the medium transfer in the Downlink
direction with a packet error probability amount of respectively $10^{-2}$, $10^{-3}$ and $10^{-4}$ 100
4.29 Throughput curves obtained by performing the long transfer in the Downlink
direction with a packet error probability amount of respectively $10^{-2}$, $10^{-3}$ and $10^{-4}$ . 101
4.30 Throughput curves obtained by performing the medium and the long transfers in
   the Uplink using 8 parallel flows ................................................................. 102
4.31 Throughput curves obtained by performing the medium and the long transfers in
   the Downlink using 8 parallel flows ................................................................. 104
4.32 Data collection of the average throughputs experienced during all of the executed
tests .................................................................................................................. 107
4.33 Data collection of the transfer times experienced during all of the executed tests . 108
4.34 Data collection showing the transfer times improvements obtained by deploying each
   PEP algorithm with respect to the case of no PEPs deployed ......................... 109
4.35 Screenshot done in the end-to-end Reno scenario with $10^{-2}$ packet error probability 111
4.36 Screenshot done in the end-to-end Reno scenario with $10^{-3}$ packet error probability 112
4.37 Screenshot done in the end-to-end Reno scenario with $10^{-4}$ packet error probability 112
4.38 Screenshot done in the PEPsal scenario with $10^{-2}$ packet error probability . . . . 113
4.39 Screenshot done in the PEPsal scenario with $10^{-3}$ packet error probability . . . . 113
List of Tables

3.1 \(<global>\) macroblock elements ......................... 40
Chapter 1

Introduction

In the world where we live today, connections are a very crucial aspect of the social life of everyone. With respect to not so many years ago, things have changed in such a way to have completely upset the sense in which we are supposed to intend the word connection itself. If we go back in time of at most ten years, the main connection services people used to ask for were mostly related to the aim of connecting each other by means of vocal calls or SMSs. Internet data application were limited to a very narrow set of mobile users, while most of the Internet traffic was dedicated to steady users. On the other hand, technology development had already started growing since years, and the existing UMTS and HSPA technologies would have been already able to support data traffic also for a larger number of users. It may probably seem weird, in fact, but the main obstacle that separated the society from the use of the Internet we are supposed to be used to since years was the fact that people still didn’t have in mind how those technologies could have been optimally exploited. The first attempts of creating killer applications gave the birth to the videochat and mobile TV services.

The real killer application idea was introduced when people started to think about the concept of smartphone. Even in this case it was not so immediate to understand the potential of this application, in fact many of historical companies that had played a leader role in the mobile phone market up to then started to disappear when everybody started to switch from the classical mobile devices to the smartphones.

Since then, things began to evolve enormously, thus heading to an exponential growth in the traffic demand and giving birth to a huge number of new applications. Nowadays, with the new mobile communication technologies (i.e. LTE, LTE-Advanced, LTE-Advanced Pro, and so on) people can be connected almost everywhere and at anytime as if they were at home, using their devices in their personal LANs. While at the beginning of the smartphones era the streaming service was mainly thought to provide such services as real-time videoconferencing, the great channel capacity
available to each of us has contributed to the development of streaming services that require a completely different type of quality of service metrics (e.g. Netflix, Instagram, YouTube are just some of the most famous platforms that in the last years have introduced the possibility to have live streaming services also in mobile devices). As most of those service providing platforms haven’t been designed for being deployed by mobile users, a great adaptation effort is required. As a consequence, today we have a huge number of issues that require to be addressed. In this thesis work, some of the solution that have been proposed so far have been analyzed and evaluated.

1.1 Objectives

The purpose of this work has been implementing and testing the performances of many different algorithms that can be adopted for the streaming of audiovisual contents in satellite networks. The implementation of the evaluated system has been developed in virtual networks using the open-source virtualization tool (VNX). The algorithms that have been evaluated in this work are intended to be used to enhance the responsivity performances of audiovisual-streaming applications, without affecting the reliability of the system. More specifically, the aim of those algorithms is to implement various kinds of FEC techniques, instead of requiring retransmissions, thus avoiding the delays typically introduced by services that are based on protocols that work in this way. The objectives can be summarized into the following points:

- PEP algorithms theoretical study.
- Filesystem configuration to suit the scenarios requirements.
- Scenarios development / Network design.
- PEP algorithms testing.

1.1.1 Requirements and Specifications

The requirements of this project can be separated into two groups:

- Theoretical requirements:
  - The TCP and UDP transport layer protocol and their algorithms.
  - Knowledges about the implemented PEP algorithms.
  - VNX language (XML based).

- Practical requirements:
  - Use of Linux.
– LXC virtual containers (virtual machines in the scenarios).
– Use of the VNX virtualization tool.
– Use of various linux tools for network simulations such as ping, netcat, traffic control.

1.2 Methods and Procedures review

The following procedures have been done during the project:

• Studying the examined PEP algorithms;
• Develop documentation about the studied algorithms;
• Installing and testing the VNX virtualization tool;
• Tune and test virtual machines (LXC);
• Develop scenarios;
• Test the algorithms in the scenarios;
• Analyze the testing results.

The first step has been studying the algorithms to be implemented. Then, for each of them a documentation that includes theoretical examples has been developed. The next step has been the setup of the environment to allow the virtual networks simulations: in this part the VNX virtualization tool and the LXC software have been installed. Then, in this part has also included the learning of how to manage and configure the containers file systems and create scenarios using the VNX XML-based language.

Once the file system is set so that all of the containers included in each scenario can both connect to each other and to the Internet, these scenarios are started and configured in different ways to test each of the studied algorithms.

1.2.1 Third-party resources

In this section, the parts that have developed the softwares used to create the virtual networks scenarios and test the algorithms are mentioned:

• VNX: Telematics Engineering Department from the Technical University of Madrid;
• LXC containers: Virtuozzo, IBM, Google, and other personal developers.
• Iperf: developed at the National Center for Supercomputing Applications at the University of Illinois by the Distributed Applications Support Team (DAST) of the National Laboratory for Applied Network Research (NLANR)
 CHAPTER 1. INTRODUCTION

• Nginx: created by Igor Sysoev and first publicly released in 2004. A company of the same name was founded in 2011 to provide support.

1.3 Thesis outline

This thesis is made up by 5 chapters.

Chapter 1 presents a short introduction about the problem statement and the overall objectives, followed by a general description of the thesis outlines and of the work flow and the methodology used to realize what presented.

Chapter 2 offers some theoretical recalls about transport-layer protocols and mechanisms and the main issues connected to them in the context of streaming applications; also, the main issues that the TCP protocol provokes in satellite networks and some of the solution so far investigated to solve them are described; finally, a quick review about the main coding theory concepts involved in the used algorithms is given.

Chapter 3 broadly explains the state of the art of the tools and softwares that have been used for realizing the simulated scenarios. Moreover, in this chapter the documentation collected about the evaluated algorithm is exposed.

Chapter 4 describes the scenario that has been set up to evaluate the studied algorithms, collects the simulation results and offers a comparison among the results obtained for each algorithm in the various experimented conditions.

Chapter 5, finally, after a short summary on the entire work, expresses some ideas about how to continue this investigation.

As just described, in the following section some information about the work flow and the methodology used to develop this thesis are provided.

1.3.1 Workflow, methodology and tools

This thesis has been developed following the following steps:

• Preliminary study on Reinforcement Learning theoretical framework;

• Study of fifth generation mobile network state of the art and design guidelines;

• Problem statement study: contextualization of the theoretical framework at the particular use case;

• Study of the simulation platform structure, gaining confidence with code and with simulation outcomes interpretation;
• Investigation about simulators output data processing, feeling comfortable using the work programming language and development environment;

• Objectives definition;

• Simulations execution and relative output data processing;

• Results interpretation and comments;

• Report writing.

The aforementioned steps haven’t been followed according to a perfectly sequential flow. The simulations have been executed using the following network testing tools: Netcat, a computer networking utility for reading from and writing to network connections using TCP or UDP; Iperf, an open-source software used for network performance measurement and tuning; Nginx, an open-source software running a web server. The simulations have been run both on the students personal laptop running Ubuntu as operating system, and on a server owned by the ENTEL department of the UPC, this one running Ubuntu Mate as operating system. In particular, the students personal computer has mainly been used to develop the example scenarios, in order to get familiar with the software to be used.
Chapter 2

Theoretical background

In this chapter a review about the theoretical concepts on which this thesis work is based is given.

2.1 Satellite communications overview

As anticipated in the previous sections, the problems that we are going to deal with throughout this thesis work are the typical ones of geostationary satellite networks. A satellite orbit is defined geostationary if the satellite always appears as a stationary object steady in the sky from the point of view of a terrestrial observer.

Satellites in geostationary orbits are far enough away from Earth that communication latency becomes significant about a quarter of a second for a trip from one ground-based transmitter to the satellite and back to another ground-based transmitter; close to half a second for a round-trip communication from one Earth station to another and then back to the first. This delay presents problems for latency-sensitive applications such as voice communication.

This work will show how to represent the features of a geostationary network within a virtual environment, and how to find solutions to the challenges that a situation like this one proposes.

2.1.1 Modeling the scenario

First of all, it is crucial to understand which are the main aspects that need to be considered to properly model a satellite network. When we think about a satellite network, we are generally brought to figure it out as a widely extended wireless link- eventually made of the combination of more and smaller wireless links- that allow the communication among hosts that may also be located in remotely far away geographical area. According to this very simple description, it is possible to make some considerations that may help us to highlight some of the main parameters about which we need to focus:
• the size aspect reminds us that, with respect to the Ethernet LANs case, we can no longer
neglect the end-to-end delay that each packet sent would experience. As we are going to see
later, this aspect is not too difficult to be handled;

• the variability of the maximum transmission rate (capacity), which is user-dependent

• the fact that we are dealing with wireless links means that the communications will also be
affected of a certain error probability, as a consequence of various phenomena as:

  – weather conditions, which may provoke the occurrence of fast-fading issue, especially
    in case of rain or high-humidity conditions;

  – population density of the area, which plays an important role in terms of interferences
    among different users;

  – shape of the covered area, as performances may also be strongly affected by the presence
    of both natural (e.g. mountains, hills, etc.) and artificial obstacles (e.g. buildings).

Moreover, in order to make the investigation easier but at the same time complete, we will consider
the channel behaviour as a time-invariant stochastic process, using different parameters combination
for every simulation, as for emulating the channel under different conditions.
2.2 Transport-layer protocols recalls

The transport layer is the fourth of the OSI/ISO protocol stack, which is the one planned to vehicle the information directly to the application-level processes running in the end users equipment. The most widely used transport-layer protocols for Internet applications are the User Datagram Protocol, UDP, and the Transport Control Protocol, TCP. Even if they have substantially the same role, the ways these protocols work are deeply different. For this reason, the choice between them strongly depends on the application they are used for.

In order to be able to perfectly understand all of the issues we will be talking about in the next sections, it’s useful to make some recalls about them.

2.2.1 The User Datagram Protocol

Defined by the RFC 768 [1], this protocol is simpler than the TCP. It’s limited to perform the multiplexing and demultiplexing functions with eventually some light error checking, thus adding no further information with respect to the IP protocol. The essential nature of this protocol is confirmed by the extremely simple format of his header, that is shown in the Figure 2.1:

The Source-Port field is the number of the port the client application process is listening to and through which it sends packets in the source host; the Destination-Port field is the number of the port the server application process is listening to and through which it sends the packets; the Length field expresses the datagram length; the Checksum field reports the result of the binary sum performed on the datagram content grouping the bits in groups of 16. It’s easy to understand that UDP cannot be a reliable and connection oriented protocol. On the same reason UDP returns to be very suitable for applications with high responsivity and low quality requirements (e.g. videoconferences, real-time streaming, etc.)

```
0    7    8    15   16    23   24    31
+----------------------------------+
|    Source   |   Destination  |
|    Port     |     Port       |
+----------------------------------+
|         |               |
|    Length  |   Checksum     |
+----------------------------------+
|               |               |
|    data octets ...          |
+-----------------------------
```

Figure 2.1: The UDP header
UDP streaming

With UDP streaming [2], the server transmits video at a rate that matches the clients video consumption rate by clocking out the video chunks over UDP at a steady rate, as UDP does not employ a congestion-control mechanism, thus being able to push packets into the network without rate restrictions.

This feature represents a first drawback of UDP streaming services because, as a consequence of the unpredictably varying amount of available bandwidth between server and client, constant-rate UDP streaming can fail to provide continuous playout. For example, in a typical scenario where every few minutes the available bandwidth drops below the video consumption rate for several seconds, a UDP streaming system that transmits video at a constant rate would provide a poor user experience, with freezing or skipped frames soon after those bandwidth falls.

For all of these problems, the video chunks transported by UDP are previously encapsulated in transport packets specially designed for transporting audio and video, using schemes like the Real-Time Transport Protocol (RTP) [RFC 3550] and Real-Time Transport Control Protocol (RTCP) [RFC 3605] or a similars. In addition, the client and server also maintain a parallel control connection over which the client sends commands (e.g. play, pause, teardown, fast forward, rewind) regarding session state changes (the Real-Time Streaming Protocol (RTSP) [RFC 2326] is a popular open protocol for this aspect), thus requiring a media control server to process client-to-server interactivity requests and to track client state for each ongoing client session. This notably increases the overall cost and complexity of deploying a large-scale video-on-demand system.

2.2.2 The Transmission Control Protocol

Defined by the RFC 793 [3], is a reliable connection-oriented protocol, that offers particular mechanisms that allow:

- to verify whether all of the sent packets are correctly arrived to the destination;
- to reorder the arrived packets directly at the receiver side;
- to adapt the bit-rate according to the channel occupation

The TCP-header format is represented in Figure 2.2. The Source and Destination Port and the Checksum fields are the same as the ones described for the UDP header; the Sequence Number field is used at the receiver side to reorder the packet before sending them to the upper levels; the Window field specifies the size of the window to be used for the flow control; the Acknowledgment Number field is used by the client to check if the receiver has correctly received the sent packets; the URG, ACK, PSH, RST, SYN and FIN flags indicate the type of information carried
### CHAPTER 2. THEORETICAL BACKGROUND

#### TCP Flow Control

To avoid losses, the Client creates a window in his transmission buffer which contains all of the packets that can be sent without being acknowledged. This mechanism is known as *Sliding Window* and is represented in figure 2.3.

When the receiver receives a packet, it controls the Sequence number field and answers to it with an ack message containing in the Acknowledgment Number field the sequence number of the oldest non-received packet. When a packet is acknowledged, the window slides until the first non-acknowledged packet. The typical way this mechanism works is showed by the sequence diagram in figure 2.4.

This mechanism is also known as *self clocking*, and allows providing an average throughput given by the ratio between the windows size (in bits) and the round trip time. As a consequence, it’s

---

**Figure 2.2: The TCP header**

```
| Source Port | Destination Port |
| Sequence Number |
| Acknowledgment Number |
| Data | |U|A|P|R|S|F| |
| Offset| Reserved |R|C|S|S|Y|I| Window |
| | |G|K|H|T|N|N| |
| Checksum | Urgent Pointer |
| Options | Padding |
| data |
```

*Figure 2.3: Representation of the Sliding-Window control mechanism*
important to control the average throughput by properly choosing the window dimension. In order to avoid the receiver buffer to get saturated, which may provoke information loss at the receiver side, this parameter is chosen according to the Advertised Window, awnd field of the last ack received. This value is reported in byte, and is calculated by the receiver as shown in Figure 2.5:

\[
awnd = RcvBuffer - \left[ LastByteRcvd - LastByteRead \right]
\]  

(2.2)

where LastByteRcvd is the last byte of the last income packet; LastByteRead is the last byte of the last packet that have already been sent to the upper layers; RcvBuffer is the overall dimension of the receiver buffer memory; awnd is the value that will be put in the Advertised Window field of the next ack sent. When the sender receives this ack, it will set the windows size W in a way to respect the following relation:

\[
W = LastByteSent - LastByteAcked < awnd
\]

(2.3)

where LastByteSent is the last byte of the last segment sent and LastByteAcked is the last byte of the last acknowledged segment.

In case of error, two situations can occur:

- if one packet gets lost, every time the receiver will receive a new message it will always answer with an ack message containing the ack number of the lost packet. By the way, as not all of
Figure 2.5: The TCP flow control mechanism
the packets follow the same path through the network, the sender will wait to receive three copies of the same ack message to be sure that the relative packet has actually been lost. When the reception of three duplicate acks (3DUPACK) occurs, the sender sends again the lost packet. The sliding-window mechanism hampers the sender to go on sending more than a certain number of packets, thus preventing the occurrence of massive losses in case of bad channel conditions. The 3DUPACK mechanism is shown in the sequence diagram in Figure 2.6;

- if the last sent packet gets lost, the 3 DUPACK mechanism wouldn’t work. For this reason, for each packet sent, the sender starts a timer so that, if something wrong happens and no 3 DUPACK is received, it is still able to detect the occurred loss comparing the timer value with the one of a previously-established Retransmission Timeout (RTO). This detection mechanism is shown in figure 2.7.

To summarize this part, we can that when using TCP a loss is detected when a 3 DUPACK is received or when a packet is not acknowledged after a time interval that is longer than the configured RTO.
TCP congestion control

It’s also crucial to give a review about the congestion control mechanism that is used by the TCP protocol. This mechanism consists in a dynamic control of the transmission rate in order to allow a maximum exploitation of the available bandwidth (which is a variable parameter that is impossible to know a priori), thus avoiding the overall network to collapse. As for the flow control, a Congestion Window, cwnd, is defined. The transmission window value is set as the minimum between the awnd and the cwnd values. The congestion control mechanism can be implemented by means of various algorithms. All of them are based on the Additive Increase Multiplicative Decrease, AIMD, paradigm. As suggested by the name, this approach envisages the use of two operations for the dynamical computation of the cwnd:

- the first operation is known as Additive Increase, as it consists in progressively increase the cwnd until a congestion event occurs. The initial value of the cwnd equals to a few MSSs (from the 2 MSSs of the initial implementaitons up to the 10 MSSs of the last implementations proposed by Google). A sstresh (slow start treshold) variable is introduced. In the first part of this phase, the cwnd is increased of 1 MSS everytime an ack is received. For this reason, this part is called slow start, and it goes on until a congestion occurs or the cwnd assumes a greater value than the sstresh variable. When a connection is started, the sstresh variable is set to a very big value, so that the first slow start only ends in case of congestion (probing
phase. In all the other cases, if the cwnd becomes greater than the sstresh variable the slow start ends and the Congestion Avoidance phase begins. In this phase, the cwnd is increased of 1/cwnd everytime an ack is received, that is 1 MSS every round trip time.

TCP has the peculiarity of being unable to distinguish congestions from other kinds of events that may cause a loss. For this reason, a congestion event is detected both in case of 3 duplicate acks reception and in case of timeout expiration.

• when a congestion occurs, the Multiplicative Decrease window reduction takes place. In this phase the cwnd value is drastically decreased in a way to prevent a collapse of the overall system. Moreover, a new value of the sstresh variable is computed. The way in which this computation is done and how the cwnd value is reduced strictly depend on the particular algorithm used.

The generical way this mechanism works is represented in Figure 2.8. The algorithm that has been implemented in this work is the Reno. This algorithm behaves in two different ways according to the type of congestion event: in case of RTO expiration, it halves the stresh value and sets the cwnd to its initial value, thus restarting with a new slow start; if the sender receives 3 duplicate acks, then the stresh values is halved again, but in this case the cwnd is set to the new sshtresh value. In this way the algorithm automatically restarts a new congestion avoidance phase when the lost packet is successfully retransmitted. This enhancement is known as fast recovery mechanism.

The overall behaviour of this algorithm can be summarized by the following pseudocode.

**Input:** sstresh

**while** ack received **do**

**if** cwnd < sstresh **then**

| cwnd ← cwnd + 1 |

**else**
\begin{align*}
cwnd & \leftarrow cwnd + 1/cwnd \\
\end{align*}

end if

do transmit_packets

end while

while 3dupack_received do
   sstresh $\leftarrow$ sstresh/2
   cwnd $\leftarrow$ sstresh
   do Retransmit_packet
   do Fast_Recovery

end while

while RTO_exp do
   sstresh $\leftarrow$ sstresh/2
   cwnd $\leftarrow$ 1
   do Retransmit_packet

end while

In HTTP streaming, the video is simply stored in an HTTP server as an ordinary file with a specific URL. When a user wants to see the video, the client establishes a TCP connection with the server and issues an HTTP GET request for that URL. The server then sends the video file, within an HTTP response message, as quickly as possible, that is, as quickly as TCP congestion control and flow control will allow. On the client side, the bytes are collected in a client application buffer. Once the number of bytes in this buffer exceeds a predetermined threshold, the client application begins playback. We just said that in that when transferring a file over TCP, the server-to-client transmission rate can vary significantly due to TCP congestion control mechanism. Furthermore, packets can also be significantly delayed due to TCPs retransmission mechanism. Because of these characteristics of TCP, the conventional wisdom in the 1990s was that video streaming would never work well over TCP. Over time, however, designers of streaming video systems learned that TCP congestion control and reliable-data transfer mechanisms do not necessarily preclude continuous playout (that’s what we are going to discuss about in the next sections) are used. The use of HTTP over TCP also allows the video to traverse firewalls and NATs more easily (which are often configured to block most UDP traffic but to allow most HTTP traffic). Streaming over HTTP also obviates the need for a media control server, such as an RTSP server, reducing the cost of a large-scale deployment over the Internet. Due to all of these advantages, most video streaming applications today including YouTube and Netflix use HTTP streaming (over TCP) as its underlying streaming protocol.
CHAPTER 2. THEORETICAL BACKGROUND

2.3 Transport-layer issues in satellite networks

GEO satellite networks will play a crucial role in future Internet due to the necessity of communication services anytime and anyplace. GEO Satellite communications are suitable for scenarios where it is difficult or impossible to deploy wired communication infrastructures. Typically these scenarios are rural environments, developing countries or airplane/boat communications. Nevertheless, GEO satellite communications have also some drawbacks. The main drawbacks are: variable link capacity, high propagation delays, packet corruption and channel asymmetry. In particular, applications that use the reliable transport protocol TCP have a significant degradation over satellite links, as its standard control algorithms are not suitable for environments where losses not only occur in case of congestion events. In this case, in fact, most of them are due to the irregularities of the wireless link, which are mainly related to the weather and other propagation conditions.

2.3.1 The Mathis equation

Before to go on with the discussion, it’s mandatory to give the following definition: by Bandwidth-Delay Product (BDP) we indicate is a parameter that quantifies the amount of data that can be transmitted over the channel without being acknowledged, that is obtained multiplying the path bandwidth by the minimum round-trip time. The importance of this parameter is given by the fact that the buffering required in a receiving system to obtain maximum performances is based on it. All of these considerations are extremely needed to envisage the throughput behaviour in satellite networks: when the TCP window size is more than the BDP, the path bandwidth represents the limit value of the throughput; but when the TCP window size is less than the load required to keep the pipe filled, the maximum throughput of the path is calculated using another model. In this case, the sending system will send a full TCP window worth of data and then waits for an acknowledgement. When the ACKs are received, more data can be sent. Therefore, the maximum throughput is the window size divided by the time it takes to get back an ACK (i.e., the RTT).

According to these considerations, we obtain a throughput value that’s considerably smaller than expected, thus proving the extremely poor TCP performances in environments that show up this feature. Nevertheless, all of these considerations assume a lossless path.

The formula that approximates what happens when TCP experiences loss in the path was approximated by Matthew Mathis [4]

\[
\text{Throughput} < \frac{\text{MSS}}{\text{RTT}} \sqrt{\frac{K}{p}}, \tag{2.4}
\]

or better

\[
\text{Throughput} = \min \left( C, \frac{\text{MSS}}{\text{RTT}} \sqrt{\frac{K}{p}} \right), \tag{2.5}
\]
where \( p \) is the packet loss probability, \( C \) the path minimum bandwidth and \( K \) a constant value that varies according to many factors, including the loss conditions of the scenario. In our scenarios we will always use the Reno as congestion control algorithm (and assuming the 3dupack case much more likely than the RTO expiration). In this situation it can be demonstrated that: in case of scenarios where losses behave as a stationary process \( K = \sqrt{1/5} \); in case of completely random losses, this constant is equal to \( \sqrt{2} \). This property can be analytically demonstrated in both cases:

**Stationary case with periodical losses** In case of periodic losses, after the slow-start phase the cwnd keeps behaving as shown in figure 2, where \( T \) is the time interval that elapses between two consecutive losses and \( M \) is assumed to be the maximum value that the cwnd can assume within a \( T \) seconds time interval (bare in mind that when the Reno congestion control algorithm is used, when a 3dupack event occurs, the cwnd is halved, and the ssthresh is set to the same value).

We can see that in the time interval between two consecutive losses the cwnd linearly varies between \( M \) and \( M/2 \), thus describing a straight segment with slope \( 1/RTT \). According to this consideration, we can easily say that

\[
\frac{1}{RTT} = \frac{M - M/2}{T} = \frac{M}{2T}.
\]

So we can say that

\[
M = \frac{2T}{RTT}.
\]

Moreover, we can also consider that:

- the loss probability \( p \) in a \( T \) seconds interval is given by the ratio between the number of losses that occur in the interval (which is 1 in our case) and the number of sent segments in the interval itself;
- we can obtain the number of segments transmitted during the interval, \( N_T \), integrating the rate in the interval itself. As the TCP rate is given by \( r(t) = cwnd(t)/\text{RTT} \), then we can also say that

\[
N_T = \int_t^{t+T} \frac{cwnd(\alpha)}{\text{RTT}} d\alpha = \frac{1}{\text{RTT}} \int_t^{t+T} cwnd(\alpha) d\alpha. \tag{2.8}
\]

The area of this integral exactly equals the area of the trapezium highlighted in Figure 2.9, so:

\[
N_T = T \frac{M + \frac{M}{2}}{2\text{RTT}}. \tag{2.9}
\]

So we can say that

\[
p = \frac{1}{N_T} = \frac{2\text{RTT}}{T (M + \frac{M}{2})}. \tag{2.10}
\]

Then

\[
T = \frac{2\text{RTT}}{p (M + \frac{M}{2})} = \frac{2\text{RTT}}{p (\frac{2M}{3\text{RTT}} + \frac{T}{\text{RTT}})} = \frac{2\text{RTT}}{p (\frac{2M}{3\text{RTT}})}.
\]

And, finally that:

\[
T = \sqrt{\frac{2}{3p}} \text{RTT} \tag{2.11}
\]

Being the throughput the ratio between the number of segments sent in a certain interval and the interval itself, we have that

\[
S = \frac{N_T}{T} = \frac{1}{pT} = \frac{1}{\text{RTT}} \sqrt{\frac{3}{2p}} \left[ \frac{\text{packets}}{s} \right] = \frac{MSS}{\text{RTT}} \sqrt{\frac{3}{2p}} \left[ \frac{\text{bits}}{s} \right], \tag{2.13}
\]

which is the same expression that we have showed before, thus demonstrating that in the steady case the \( K \) constant equals to \( \sqrt{1.5} \).

**Stationary case with random losses** As regards the second case, if we consider that, according to the Reno algorithm, the cwnd varies every round trip time as follows:

\[
\Delta cwnd = \begin{cases} 
\frac{1}{cwnd}, & \text{if an ack is received} \\
-\frac{cwnd}{2}, & \text{if a loss is detected}
\end{cases} \tag{2.14}
\]

So, the average \( cwnd \) variation is given by

\[
\Delta cwnd = (1-p) \frac{1}{cwnd} + \left( -\frac{cwnd}{2} \right) = 0 \tag{2.15}
\]

\[
(1-p) \frac{1}{cwnd} = p \frac{cwnd}{2} \Rightarrow cwnd^2 = \frac{2}{(1-p)p} \Rightarrow cwnd = \sqrt{\frac{2}{(1-p)p}} \tag{2.16}
\]

Assuming that losses probability is much smaller than 1, we can finally say that:

\[
cwnd \approx \sqrt{\frac{2}{p}} \tag{2.17}
\]

which is the same expression that we have showed before, thus demonstrating that in the random losses case the \( K \) constant equals to \( \sqrt{2} \).

As a final comment about these relations, it is possible to state that both of them show how quickly TCP performances go down increasing the packet loss introduced by the scenario.
2.3.2 TCP improved schemes for wireless lossy networks

Reliable transport protocols such as TCP are tuned to perform well in traditional networks where packet losses occur mostly because of congestion. However, networks with wireless and other lossy links also suffer from significant non-congestion-related losses due to reasons such as bit errors and handoffs. TCP responds to all losses by invoking congestion control algorithms, resulting in degraded end-to-end performance in wireless and lossy systems. In this section we intend compare several schemes designed to improve the performance of TCP in such networks. These schemes are classified into three broad categories [5] end-to-end protocols, where the sender is aware of the wireless link; link-layer protocols, that provide local reliability; and split-connection protocols, that break the end-to-end connection into two parts at the base station.

End-To-End Schemes

The current de facto standard for TCP implementations is TCP-Reno (Figure 2.10). TCP-New Reno protocol improves the performance of TCP-Reno after multiple packet losses in a window by remaining in fast recovery mode if the first new acknowledgment received after a fast retransmission is less than the value of the last byte transmitted when the fast retransmission was done (Figure 2.11).

The TCP+SACK protocol adds selective acknowledgments to the standard TCP Reno stack. This allows the sender to handle multiple losses within a window of outstanding data more efficiently. However, the sender still assumes that losses are a result of congestion and invokes congestion control procedures, such as shrinking its congestion window size. This allows us to
identify what percentage of the end-to-end performance degradation is associated with standard TCPs handling of error detection and retransmission.

**Link-Layer Schemes**

Existing link-layer protocols choose from techniques such as Stop-and-Wait, Go-Back-N, Selective Repeat and Forward Error Correction to provide reliability. A base link-layer algorithm, called LL (Link-Layer, indeed), uses cumulative acknowledgments to determine lost packets that are retransmitted locally from the base station to the mobile host. To minimize overhead, an improved implementation of LL leverages off existing TCP acknowledgments instead of generating its own. Timeout-based retransmissions are done by maintaining a smoothed round-trip time estimate, with a minimum timeout granularity of 200 ms to limit the overhead of processing timer events. This still allows the LL scheme to retransmit packets several times before a TCP-Reno transmitter would timeout.

**Split-Connection Schemes**

A split scheme uses one (or more) intermediate host to divide a TCP connection into two separate TCP connections. The objective of splitting is isolating the long-latency satellite link with middle agents called Performance-Enhancing Proxies (PEPs). With splitting, the connection is divided in three connections: sender-to-PEP, PEP-to-PEP and PEP-to-receiver. As we will see later, this approach also allows to use different protocols according to the features of each of the branches in which the link is divided. In the next chapter we will try to prove the effectiveness of this approach, also comparing among them many different implementations of this scheme.
CHAPTER 2. THEORETICAL BACKGROUND

2.4 FEC techniques

As said in the latest lines, the approach we are going to implement in this work is the one based on the TCP split model. This scheme proposes to divide one single TCP connection in many branches where different protocols can be used, as a solution to solve the problems connected to the low performances that TCP shows up in environments like the satellite networks, where large end-to-end delays and relatively high packet loss rates occur. As just described, it envisages the implementations of two Performance-Enhancing Proxies (PEPs) to be put at the boundaries of the lossy branch, and which use a different communication protocol to communicate between them, in order to overcome the aforementioned problems, thus enhancing the overall quality of the end-to-end communication. Giving a look at various of the studies that have been done in this field, it is possible to notice that a lot of the proposed solutions for the PEP-to-PEP communication problem consist of implementing some modifications to the transport-layer protocols and techniques adopted. More specifically, the algorithms that we are going to study and evaluate in this work are based on the Forward Error Correction (FEC) technique.

The FEC technique allows dealing with the problems introduced by losses including in the sent packets some redundancy that allow to correctly reconstruct the data even in case of losses or errors. This means that in these packets the information needs to be codified in a way to enable the receiver to detect and eventually correct errors. The codes that allow these protection mechanisms are known as error-detecting and error-correcting codes.

In an error correcting code, a message \( M \) is encoded as a sequence of symbols known as codeword. The set of possible symbols is fixed in advance. The code incorporates some redundancy, so that if some of the symbols in a codeword are changed, we can still figure out what the original message must have been. The simplest example of an error correcting code is the triple-redundancy code, in which the message consisting of a single symbol \( a \) is encoded by repeating the symbol three times. Suppose one symbol in the codeword is changed, so we receive a word \( baa \) or \( aba \) or \( aab \). We can still recover the original symbol \( a \) by taking a majority vote of the three symbols received. If errors result in two symbols being changed, we can not correct the errors and recover the original message, but we can at least detect the fact that errors occurred. By repeating the message more times, we can achieve higher rates of error correction: with five repetitions we can correct two errors per codeword, and with seven repetitions we can correct three. By the way, these simple-minded error correcting codes are very inefficient, however, since we have to store or transmit three, five or seven times the amount of data in the message itself. Using more sophisticated mathematical techniques, we will be able to achieve greater efficiency.
2.4.1 Network Coding

A more sophisticated coding technology is the Network Coding [6]. This technique is also the one on which one of the algorithms we have tested is based.

According to this technique, when a user needs to transfer a data block \( D \) over the network the following steps are performed:

- First, the block is divided into \( n \) segments \( S_i \) of the same length, \( D = [S_1, \ldots, S_n] \);

- Then, the node computes \( n \) new segments by performing a random linear combination of each of the segments that composed the original block. More specifically, each segment is interpreted as a vector of symbols over the field \( \mathbf{GF}(2^q) \), where \( q \) is the number of bits that make up one symbol.

- The coded payload is constructed by randomly selecting coding coefficients \( C_i = [C^1_i, \ldots, C^n_i] \), and computing:

\[
P_i = C_i \cdot D = \sum_{k=1}^{n} (c^k_i \cdot S_k);
\]

- Finally, each coded packet will be given by a payload \( P_i \) and additional metadata.

When \( n \) linearly independent coded packets arrive to the receiver, it can decode the original data \( D \) by solving the following system of \( n \) equations and \( n \) unknowns:

\[
\begin{bmatrix}
c^1_1 & c^1_2 & c^1_3 & \ldots & c^1_n \\
c^2_1 & c^2_2 & c^2_3 & \ldots & c^2_n \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
c^n_1 & c^n_2 & c^n_3 & \ldots & c^n_n
\end{bmatrix}
\begin{bmatrix}
S_1 \\
S_2 \\
\vdots \\
S_n
\end{bmatrix}
= 
\begin{bmatrix}
P_1 \\
P_2 \\
\vdots \\
P_n
\end{bmatrix}
\]  

The dot and matrix products computed over \( \mathbf{GF}(2^q) \) described by the last equation may be of significant computational cost, thus representing a Network Coding drawback. On the other hand, the main advantages connected to the implementation of this technique result to be the following ones:

- it is possible to generate infinitely many coded packets starting from a finite set of segments. As the linear equations used to generate the payloads are linearly independent, given any set of \( n \) coded packets, it is possible to decode \( D \) with high probability with high probability, once \( n \) degrees of freedom arrive at the receiver. For this reason, in case of losses the sender only needs to construct as many packets as those lost and resend these. With high probability, the additional packets will be linearly independent with the packets sent thus far, and therefore be helpful towards the completion of the data transfer. This property is known as Rateless Erasure Codes;
• The described scheme is oblivious to the source where the degrees of freedom originated. Hence, an arbitrary number of senders serving the same data D could be used to independently produce the necessary degrees of freedom to complete a data transfer. The latter is done asynchronously, as the payloads generated at each of these sources are linearly independent with high probability, and no synchronization is needed between the different sources for this property to hold. This property is known as Near-Perfect Asynchronous Coordination.

The Network-Coded TCP (CTCP)  In this paragraph, we describe the design and implementation of CTCP, a novel TCP-like (i.e. reliable) protocol that uses network coding. This protocol uses the Sparse Network Coding (SNC), which is a method that reduces the computational complexity of classic network coding by restricting the way in which the coding coefficients $C_i$ are chosen at the sender, thus constraining the set of possible matrices which need to be inverted to decode a given data block and reducing the number of entries that need to be computed in order to find $C^{-1}$. One way to accomplish this would be to place restrictions so that both $C$ and $C^{-1}$ are sparse\(^1\). SNC uses a family of sparse matrices called banded matrices, in which non-zero elements lie near the diagonal. This property allows implementing a more efficient Gaussian elimination process. The main idea behind SNC is based on the following empirical finding which states that if $C$ is a banded matrix with small bandwidth (roughly $O(\log n)$) then $C^{-1}$ is sparse with high probability. Restricting the way the $C_i$’s are chosen, it is possible to make sure that $C$ is a banded matrix. Namely, these coefficients are chosen so that the matrix has at most $k$ consecutive non-zero coefficients. In this way, each coded packet will be constructed by computing a random linear combination of at most $k$ consecutive segments.

Besides this coding aspect, the CTCP protocol also differs with respect to the classic TCP from other implementation points of view. These features make this protocol particularly suitable for scenarios as satellite networks, in which delay sensitive applications turn to be penalized. As an example, while the baseline TCP implementation (Reno) uses losses as an indication of network congestion and subsequently back its sending window off, the CTCP algorithm uses a variation on the Vegas algorithm, in which the congestion window is calculated as a function of the RTT delay. We will talk about this aspect in the next chapter, when describing the TCPeP algorithm, which is based on a particular implementation of the CTCP.

2.4.2 Reed-Solomon

Reed-Solomon codes [7] are error correcting codes, in which redundant information is added to data so that it can be recovered reliably despite errors in transmission. These codes are used for example for the error correction system used on CDs and DVDs, and also on satellite links.\(^1\)

\(^1\)In numerical analysis, a sparse matrix is a matrix in which most of the elements are zero.
According to codes theory, linear codes offer the advantage of being easy to encode. On the other hand, it can be demonstrated that the inverse operation is not always so easy to do, nor the weight is so easy to be computed. For this reason linear codes that also have some algebraic properties that make those tasks easier are often chosen for these purposes. This class of codes is known as algebraic codes. The Reed-Solomon codes are particular algebraic codes defined as follows:

**Definition 1.** Let \( p \) be a prime number and let \( m \leq n \leq p \). The Reed-Solomon code over the field \( \mathbb{Z}_p \) with \( m \) message symbols and \( n \) code symbols is defined as follows. Given a message vector \([x_1, x_2, \ldots, x_m]\), let \( P(t) \) be the polynomial

\[
P(t) = x_m t^{m-1} + x_{m-1} t^{m-2} + \ldots + x_2 t + x_1
\]

with coefficients given by the message symbols. Thus \( P(t) \) is a polynomial of degree at most \( m - 1 \) in one variable \( t \), with coefficients in \( \mathbb{Z}_p \). Then the code vector \( a \) for this message vector is the list of the first \( n \) values of the polynomial \( P(t) \):

\[
a = [a_1, a_2, \ldots, a_n] = [P(0), P(1), \ldots, P(n-1)]
\]

(evaluated using modular arithmetic in \( \mathbb{Z}_p \)).

**Theorem 1.** The Reed-Solomon code over \( \mathbb{Z}_p \) with \( m \) message symbols and \( n \) code symbols is the linear code with matrix

\[
C = \begin{bmatrix}
1 & 1 & 1 & \ldots & 1 \\
0 & 1 & 2 & \ldots & n-1 \\
0^2 & 1^2 & 2^2 & \ldots & (n-1)^2 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0^{m-1} & 1^{m-1} & 2^{m-1} & \ldots & (n-1)^{m-1}
\end{bmatrix}
\]

all entries taken (mod \( p \)).

It is able to detect and correct multiple symbol errors. By adding \( t \) check symbols to the data, a Reed-Solomon code can detect any combination of up to \( t \) erroneous symbols, or correct up to \( t/2 \) symbols. Reed-Solomon codes are also suitable as multiple-burst bit-error correcting codes, since a sequence of \( b + 1 \) consecutive bit errors can affect at most two symbols of size \( b \). The choice of \( t \) is up to the designer of the code, and may be selected within wide limits.
Chapter 3

State of the art of the used technologies

The aim of the following sections is to give a description of the software and the algorithms that have been used and studied in this work.

3.1 Used software

In this section, the software used in this project are described.

3.1.1 LXC

Linux Containers (LXC) is a virtualization tool, that works at operating-system level to allow running many Linux systems over a single monolithic Linux kernel from a Linux host. Moreover, LXC provides cgroups functionality, which isolate the resource usage of the virtual containers and isolated namespaces, thus providing a virtual isolated environment for processes. In order to better understand how these containers can be configured, it’s important to get acquainted about what happens when a new Linux container comes to be instantiated. Figure 3.1 will be used as a reference for this explanation. When the LXC software is installed on the localhost, a virtual bridge named lxcbr0 is instantiated. This is the interface that will be used to connect a container to the eth0 interface of the localhost once the container itself is created. Then, every time a new LXC container is created, a new virtual interface is instantiated on this virtual router, in a way to realize a link between the two elements. The interfaces created in the container and the virtual bridge realize a particular association between these entities, which is known as Virtual Ethernet (Veth) Adapter. From a network configuration point of view, Veth adapters are not recognized in the same way both in the container and in the virtual bridge: while the container recognizes that
interface as its own eth0 interface, the physical host directly recognizes the association between that virtual interface and the container itself. This aspect will be better specified when we will take a look at the configuration file. To create, start and attach to a new container the commands to be used are the following ones:

```
lxc-create -n <container_name> -t ubuntu
lxc-start -n <container_name>
lxc-attach -n <container_name>
```

Once the container has been created, it is possible to modify its networking configuration by editing the following file in the physical host file system:

```
var/lib/lxc/container_name/config
```

This file allows the declaration and definition of new network interfaces, which configuration can be freely customized in any moment, by simply adding or modifying some parameters. In our case, the parameters used for our configurations are the ones listed in the example below:

```
lxc.network.type=veth
```

indicates that the virtual interface that we are creating behaves like an ethernet interface;

```
lxc.network.veth.pair=pairname
```

allows us to give a name to the Veth Adapter associated to the interface (e.g. if the device is named pep and this is its eth0 interface, an typical way to name the parameter may be pep-e0);

```
lxc.network.name=ifname
```

indicates the name of the interface associated to the adapter by the container (e.g. eth0);

```
```
indicates the MAC address of the container;

\verb+lxc.network.link=virtualbridgenname+

finally indicates the virtual bridge to which the container is attached through this interfaces.
Moreover, it is also possible to add other brand-new virtual bridges, e.g. to allow connection between containers. We can also see that a NAT-masquerading operation is needed to be applied in order to allow the ip packets sent by a container to be forwarded into the host LAN, thus allowing connection with other hosts, even outside the LAN itself.

In the end, it\textquotesingle s important to specify that a container is not at all a complete virtual machine ready to work for our scopes. In fact, when create it needs the following actions to be performed in order to make it completely usable: First of all, it needs a default gateway to be indicated, in order to be able to also send packets outside its own LAN. We will see in the next section that in our case this in made possible thanks to the VNX software, which assigns an IP address to the lxcbr0 virtual bridge and performs the masquerading operation for the created containers. This operation is essential for doing many things, e.g. accessing the Internet. The command that allows us to do this is: route add default gw xx.xx.xx.xx After this, to allow the address resolution by means of a certain nameserver, the following command is used:

\verb+echo "nameserver 8.8.8.8" > /etc/resolv.conf+

In our case, the nameserver associated to the specified IP address is the Google public DNS. This couple of operations allows us to perform the following basic settings:

\verbatim
[sudo] apt-get update
[sudo] apt-get install git
[sudo] apt-get install build-essential
\endverbatim

Once done this, we can continue installing any kind of program we will need (iptables, iperf, nginx, etc.).

3.1.2 VNX

A very important tool used in this work is Virtual Networks over Linux (VNX). It is an open-source virtualization tool, that allows creating virtual scenarios of virtual machines defined by the user using an xml based language. Once the scenario is defined in a .xml file, the software parses the code and automatically generates the virtual scenario. In this way, VNX allows to simulate networks avoiding the efforts that would be required testing the networks in real equipment in terms of hardware, time and, obviously, money. One of the improvements that VNX includes with respect to its predecessor (VNUML) is the capability to use LXC containers as virtual machines.
This explains can help to give a flavor about how powerful this software can be. Once we have in mind how to configure our scenario, we only need to write its configuration in a xml document.

**VNX basic commands**

Once described the scenario in this configuration document, it can be created by simply typing the following commands:

```bash
[sudo] vnx -f /path/to/the/scenario/file.xml --create
[sudo] vnx -f /path/to/the/scenario/file.xml --start
```

Once done this, the scenario will be created automatically, and the features of the new elements introduced can be controlled with the `ifconfig` and `brctl show` commands. In this way it will also be possible to control the configuration of the created containers checking the folder `/var/lib/lxc`. Finally, it is also possible to generate a picture representing the created topology using the command:

```bash
[sudo] vnx -f /path/to/the/scenario/file.xml --show-map [svg|png]
```

Other commands we have been using during this work are the following ones:

- **[sudo] vnx -f /path/to/the/scenario/file.xml [-M vm_name] -d|--shutdown**
  
  used to shut down one or all of the containers of the scenario;

- **[sudo] vnx -f /path/to/the/scenario/file.xml -P|--destroy**

  used to destroy the whole scenario;

- **[sudo] vnx -f /path/to/the/scenario/file.xml [-M vm_name] -x|--execute|--exe command_label**

  used to execute some particular command defined in the configuration file;

- **[sudo] vnx -f /path/to/the/scenario/file.xml --exe-info**

  used to show info about the commands defined in the configuration file under the `<exec>` and the `<filetree>` labels.

**Configuration document structure**

The scenario configuration document is written using the xml metalanguage. The document template is organized in many different sections, all of which are included in the principal node, indicated by the following tags [8]:
CHAPTER 3. STATE OF THE ART OF THE USED TECHNOLOGIES

<table>
<thead>
<tr>
<th>Element</th>
<th>Mandatory/Optional</th>
<th>Unique/Multiple</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;version&gt;</td>
<td>Mandatory</td>
<td>Unique</td>
<td>VNX specification language version</td>
</tr>
<tr>
<td>&lt;scenario_name&gt;</td>
<td>Mandatory</td>
<td>Unique</td>
<td>Scenario name</td>
</tr>
<tr>
<td>&lt;ssh_version&gt;</td>
<td>Optional</td>
<td>Unique</td>
<td>SSH version to connect to the VMs</td>
</tr>
<tr>
<td>&lt;ssh_key&gt;</td>
<td>Optional</td>
<td>Multiple</td>
<td>Host directory containing the SSH key</td>
</tr>
<tr>
<td>&lt;Automac/&gt;</td>
<td>Optional</td>
<td>Unique</td>
<td>Allows VMs MACs自动 generation</td>
</tr>
<tr>
<td>&lt;vm_mgmt&gt;</td>
<td>Mandatory</td>
<td>Unique</td>
<td>Allows configuring further options</td>
</tr>
<tr>
<td>&lt;vm_defaults/&gt;</td>
<td>Optional</td>
<td>Unique</td>
<td>VMs parameters default values</td>
</tr>
</tbody>
</table>

Table 3.1: <global> macroblock elements

These sections are organized as to form four macroblocks. Not all of them are mandatory to be included. Some of these macroblocks can group more similar sections. The parameters of each section are declared as elements of the section. Each macroblock has its types of elements. Some of these elements are mandatory, others are optional. In the following tables, we give a brief description of the elements that have been used in this work for each macroblock.

**The <global> macroblock** It’s a mandatory macroblock composed of an unique section, that groups the general elements that characterize the whole scenario. In our case we will always set these elements and their attribute as follows:

```xml
<global>
    <version>2.0</version>
    <scenario_name>lorenzo4</scenario_name>
    <ssh_version>2</ssh_version>
    <ssh_key>~/.ssh/id_rsa.pub</ssh_key>
    <automac/>
    <vm_mgmt type="net" network="172.16.31.0" mask="24" offset="10">  
        <mgmt_net net="lxcbr0" managed="no" config="manual"/>
    </vm_mgmt>
    <vm_defaults/>
        <console id="0" display="no"/>
        <console id="1" display="yes"/>
</global>
```

```xml
<vnx>
...
</vnx>
```
The `<net>` macroblock  It’s an optional macroblock that can be composed of many similar sections, each of which defines a network of the scenario. Every instance of this macroblock leads to the creation of a network device that links to each other all of the machines that belong to that network, and takes the name of the network itself (which is a mandatory attribute that every `<net>` instance needs to have). The type of the network device depends on which value is specified in the mode attribute (also this one is mandatory for every instance). If the value `virtual_bridge` is indicated, then a virtual bridge is created, as it would happen using the command `brctl add name_br` (and this will be the choice we will do) in the host; otherwise, if the value `uml_switch` is indicated, the network will be used as for a `uml_switch` process executed in the host. The syntax to be adopted for this section is like that described in the following line:

```
<net name="network_name" mode="virtual_bridge" />
```

The `<vm>` macroblock  It’s an optional macroblock that can be composed of many similar sections, each of which defines the parameters that characterize one of the virtual machines of the scenario. It is featured by two mandatory attributes (name and type), plus more optional attributes that we are not going to use, and a number of optional elements. We briefly introduce which are the elements that have been used to configure the virtual machines instanciated in our scenarios:

- **filesystem**: defines the file system used for the VM. This element has one mandatory attribute, that specifies the type of file system that we are going to use for the VM;

- **on_boot**: indicates the sequence of commands to be executed when the VM is started;

- **if**: defines one interface of the VM. In this element it is also possible to instantiate an `ipv4` element, to specify the IPv4 address to associate to this interface;

- **route**: this field allows configuring the routing tables to reach some specific networks, indicating the type of network-level protocol adopted (IPv4 in our case) and the gateway IP address;

- **forwarding**: activates the IP forwarding between the interfaces of the VM;

- **filetree**: specifies a directory in the host file system to be copied in the VM when the command expressed in the seq attribute is executed;

- **exec**: specifies a command to be executed by the VM.
Figure 3.2: Summarization of the structure of a VNX configuration file

The `<host>` macroblock It’s an optional macroblock composed of a unique section, in which parameters to be implemented by the host while the scenario is running are defined. The elements that we are going to use in this work are `hostif`, `route`, `forwarding` and `exec`, and work in the same way of the `if`, `route`, `forwarding` and `exec` elements defined for the previous section. Finally, in the scheme in Figure 3.2 we can find a scheme that summarizes the overall structure of a VNX configuration file. Note that the elements are reported in black rectangles while those reported in the green ones are attributes.
Example scenario

As an example, in Figure 3.3 we report a very simple scenario as an example to get familiar with the fundamental features of this software.

As we can see from this picture, the scenario is given by two hosts and a bridge that interconnects them. More specifically, the host in green represents the physical host, the one in light blue is a LXC container (as stated by the LXC acronymous under its name), and the bridge is a virtual bridge (vbd). In the global macroblock, we find the following declarations:

- in the `vm_mgmt` element:

  ```xml
  <vm_mgmt type="net" network="172.16.31.0" mask="24" offset="10">
    <mgmt_net net="lxcbr0" managed="no" config="manual"/>
  </vm_mgmt>
  ```

  These attribute values state as follows:

  - `type="net"` sets a device as the one to which all of the VMs interfaces, as well as the host interface, are connected, as a *UML switch*, to which is also possible to assign an IP address. This is a very important feature, as it both allows us to create a LAN among all of the VMs of the scenario, and to use this device as a gateway that allows these VMs to communicate with external networks (e.g. the Internet);
  - `network="172.16.31.0"` sets the ip address of the LAN;
  - `mask="24"` sets the netmask of the LAN.

Moreover, in this case (type="net") a `mgmt_net` element is also mandatory to be declared. This element is set to manage the *uml_switch* LAN. Its attributes are set as follows:
– net="lxcbr0" indicates the name of the device to be used as UML switch;
– managed="no" means that the IP addresses of the VMs in this LAN are automatically assigned by the software;
– config="manual".

- in the vm_defaults element:

  <vm_defaults>
  <console id="0" display="no"/>
  <console id="1" display="yes"/>
  </vm_defaults>

  The console element allows us to set information about the console where to start the VMs.

As said in the previous section, the elements that appear in the network are defined in the net and vm macroblocks (which, we remind, can be composed of more than one section). The net macroblock reports the same syntax described in the previous paragraph. After that, we find the vm macroblock, where the remote machine is defined as follows:

  ...
  <vm name="remote" type="lxc" arch="x86_64">
  <filesystem type="cow">/usr/share/vnx/filesystems/rootfs_lxcUbuntu64</filesystem>
  <if id="1" net="vbridge">
    <ipv4>192.168.1.2/24</ipv4>
  </if>
  <route type="ipv4" gw="172.16.31.1">default</route>
  ...

In the first line we define the name and the type of virtual machine, that obviously is LXC, and the architecture. Then the filesystem and the interfaces ip addresses are instanciated. Better specifying, the filesystem type="cow" expression means that the VM filesystem will be created starting from the indicated file system by means of a copy-on-write operation, so that the original one doesn’t report changes. Moreover, we also notice that we can directly define in this file the default gateway. At this point, typing the ifconfig command in the physical host we see that the situation is the following one

ifconfig

eth0 Link encap:Ethernet  HWaddr d2:24:27:1a:00:aa
From a software point of view, the interconnections created among the different objects defined in this scenario can be summarized as in the scheme of Figure 3.4. If we use the vbridge network and try to communicate with the local host through the vbridge everything will work fine, as shown in the screenshots reported in figure 3.5, in which we report the packet exchanged during a complete netcat conversation between the two hosts captured through the vbridge network device.
### 3.1.3 Iperf

Iperf is an open-source and cross-platform tool written in C and widely used for network performance measurement and tuning. It has client and server functionality, and can create data streams to measure the throughput between the two ends in one or both directions. Typical Iperf output contains a time-stamped report of the amount of data transferred and the throughput measured reporting information about:

- **Latency (response time or RTT):** can be measured with the Ping command.
- **Jitter (latency variation):** can be measured with an Iperf UDP test.
- **Datagram loss:** can be measured with an Iperf UDP test.

The data streams can be either Transmission Control Protocol (TCP) or User Datagram Protocol (UDP):

- **UDP:** When used for testing UDP capacity, Iperf allows the user to specify the datagram size and provides results for the datagram throughput and the packet loss.
- **TCP:** When used for testing TCP capacity, Iperf measures the throughput of the payload.
The TCP protocol is used by default. Jperf is a tool written in Java that can be associated with Iperf to display the results in a graphical frontend.

Finally, iperf is a very important analysis tool also thanks to its high simplicity. In fact, the commands needed to perform a complete test that may also contemplate the transfer of a file are the following ones:

client side: `iperf -c <server_ip> [-u|--udp] [-F|--fileinput filename]` ;
server side: `iperf -s` .

3.1.4 NGINX

NGINX is a free, open-source HTTP server and reverse proxy, well recognized for its high performances, stability, rich feature set, simple configuration, and low resource consumption. Nginx was written with the goal of outperforming the Apache web server as Nginx uses dramatically serving static files. This approach allows using much less memory than Apache, and handling roughly four times more requests per second. This performance boost comes at a cost of decreased flexibility.

Thanks to all of these features, several high-visibility sites (e.g. Netflix, Wikipedia, Pinterest, Airbnb, WordPress.com, GitHub, etc.) are powered by NGINX.

In our scenarios, we have implemented this software in the remote host virtual machine. To run the nginx software in the remote host virtual machine, all we had to do was to execute the following command in the local host root terminal:

```
vnx -f /path/to/the/scenario/file.xml --execute nginx_start -M remotehost
```

According to the xml code reported in Appendix A.3, this command starts the nginx server application in the remote host virtual machine. It is possible to check that the software is correctly working in the remote VM by simply typing in its terminal the command:

```
root@remotehost:~# netstat -antp
```

If everything has been set properly, the following message should then be prompted.

<table>
<thead>
<tr>
<th>Proto</th>
<th>Recib</th>
<th>Enviad</th>
<th>Direccin local</th>
<th>Direccin remota</th>
<th>Estado</th>
<th>PID/Progr am name</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcp</td>
<td>0</td>
<td>0</td>
<td>0.0.0.0:80</td>
<td>0.0.0.0:*</td>
<td>ESCUCHAR</td>
<td>515/nginx</td>
</tr>
</tbody>
</table>

This message shows that the nginx server is running and listening on the local TCP port 80. Once this simple procedure has been done, it is possible to browse any file that is stored in the server VM by simply typing the following command in the local host terminal:

```
local@localhost:$ browser_application http://<server_IP>/file.extension
```
CHAPTER 3. STATE OF THE ART OF THE USED TECHNOLOGIES

3.2 Studied Algorithms

The aim of this section is to give an explanation about how the studied and used algorithms work.

3.2.1 UDPspeeder and tinyfecVPN

The UDPspeeder algorithm provides a solution that is based on settling a tunnel that improves the network quality on a high-latency lossy links by using Forward Error Correction.  

This algorithm can be both be used alone and combined to other network tool that provide transport-layer encapsulation functionalities. In fact, as the name would suggest, this software is designed to improve the performances of communications carried on the UDP transport protocol. Using a tool that perform UDP encapsulation of the incoming packets allows exploiting its performance-improving capabilities also in cases in which, for example, the TCP protocol is used. As the one just described is actually the situation in which we are supposed to be in the cases that we want to evaluate, a quick description about some tools that allow this tunnelization will be also given.

First, let’s focus about the UDPspeeder algorithm itself. Let’s assume that the link from our local network to the remote server is lossy, and that we are communicating using UDP as transport layer protocol. With well-tuned parameters, UDPspeeder allows to easily reduce UDP packet-loss-rate to less than 0.01 percent using a particular FEC technique which is based on the Reed-Solomon code. Obviously, it provokes an increased expense in terms of bandwidth.

As previously reported, the algorithm is supposed to be deployed over lossy, high-latency links, such as the satellite links. The simplest idea to set the environment is based on the splitted approach. This requires that in both the local and the remote LAN a node is selected to work as a PEP. The idea on which this approach is based is summarized by the scheme in Figure 3.6.

After having downloaded and compiled the code (https://github.com/wangyu-/UDPspeeder.git) in both the proxies, the algorithm is run by typing the following commands:

```
root@localpep:~# ./speederv2 -c -l 0.0.0.0:7777 -r remote_proxy_IP:4096 -f20:10
-k "passwd"
```

Figure 3.6: Schematic representation of the behaviour of the UDPspeeder algorithm
The syntax of those commands, which basically run the speederv2.sh using the specified parameters, can be explained as follows (bare in mind that we are analyzing, as an example, the case of a communication that take place in the direction from the local to the remote node).

- the application running in the local proxy listens to its own UDP port 7777. The "-c" tag indicates that this proxy will act as client proxy, so that its functions will be appending the FEC tailer to the packet and converting the destination port value (in this case to 4096) of the packet itself. Once these functions are executed, the packet is then routed to the destination proxy;

- the application running in the remote proxy listens to its own UDP port 4096. The "-s" tag indicates that this proxy will act as server proxy, so that its functions will be checking the correctness of the received packet, remove the FEC and modify the destination port. Once these functions are executed, the packet is finally routed to the destination node.

- the "-k" parameter indicates a string that is used as key for simple xor encryption. If not set, then the encryption is disabled;

- the -f parameter indicates that the FEC technique is set in a way to send 10 redundant packets for every 20 sent packets. This is the parameter that at most can be set according to the specific speed and quality requirements.

The approach described so far fits well in case of end-to-end UDP communications, but the use case where we are going to deploy and evaluate this algorithm the TCP is the transport protocol adopted by the remote nodes. By the way, forecasting the problems that may arise in these situations, the author of the algorithm also proposes a solution to make it useable also when the TCP or ICMP protocols are used above the network layer. This solution consists in encapsulating the packets exchanged through the two PEPs, thus building up an UDP tunnel, that is an overlay network by means of which the PEPs communicate in a totally transparent way with respect to the nodes between them. The idea this approach lays on is summarized by the scheme in Figures 3.7 and 3.8.

Currently, OpenVPN, L2TP, ShadowVPN are the tools able to perform this tunnelling function and, at the same time, that are confirmed to work well with this algorithm.

An alternative way to implement the same mechanism using a single software is to deploy the tinyfecVPN algorithm, which realizes a VPN with Build-in Forward Error Correction(FEC) Support. tinyfecVPN uses same lib as UDPspeeder and supports all FEC features of UDPspeeder.
CHAPTER 3. STATE OF THE ART OF THE USED TECHNOLOGIES

Figure 3.7: Schematic representation of the behaviour of the UDPspeeder algorithm over an UDP tunnel

Figure 3.8: Schematization of the FEC code appending and UDP tunnelling processes
Even if both of these algorithms have been analyzed in details, they haven’t been deployed and evaluated over the created virtual scenario, as the implementation of both the combined scheme (UDPspeeder over UDP tunnel) and the integrated one (tinyFEC) have given us many problems to be run. Moreover, we didn’t expect many improvement from this scheme, as the single implementation of FEC performs good in terms of error correction, but decreases the obtained speedth expcially when used in combination with UDP and IPsec encapsulations (also because it has been mainly designed to improve UDP-based connection rather than TCP ones). For these reasons, it has been preferred to go on evaluating the other studied algorithms, keeping in mind that using the FEC technique may still be a way to obtain important performance improvements, but only if implemented in a more efficient way, as will be shown in the next sections.

3.2.2 PEPsal

PEPsal is a multi-layer proxy algorithm. At the network level, it uses netfilter and iptables to intercept those connections that would involve a satellite links and "steals" the TCP SYN packet in the three-way handshake phase of a TCP connection, then pretend to be the other side of that connection, and initiate a new connection to the real endpoint, using a userspace application that directly copy data between the two sockets, thus implementing the SPLIT-connection approach as well as the previous algorithms. This split is transparent at transport layer, as modifications are not needed in both of the connection endpoints. For this reason, the TCP user doesn’t even realize about the connection splitting.

By the way, this approach leads to some security issues that actually compromise the suitability of this algorithm in contexts that impose strict security constraints:

- as indicated in Figure 3.9 by the green arrows, in order to realize the TCP split, PEPsal performs premature acknowledgement of data. This approach allows reducing the influence of high latency, but also makes impossible guaranteeing that end-to-end reliability;

- since PEPsal violates the end-to-end semantics, it automatically makes impossible the end-to-end use of IPsec, which requires the protected information to remain immuted throughout all the secure path.

Besides the TCP split, in order to obtain performance enhancements the PEPsal algorithm also leverages on implementing aggressive congestion avoidance algorithms, in a way to allow reestablishing the communication in a faster way in case of losses occurrence. More expically, the congestion avoidance algorithm suggested in the article that can be found at the GitHub link where it is possible to download the code [9] is the TCP Hybla [10], which has been designed upon an analytic evaluation of TCP congestion window behaviour in the TCP NerReno algorithm, with the goal to be implemented in heterogeneous networks, in order to cope with the same problems we
are dealing with in this work (performance enhancement over high-latency lossy links). However, we will see in the next chapter that the software allows us to freely configure the used congestion avoidance algorithm by simply modifying a parameter in one of the PEPsal configuration files. If combined with the TCP split approach, this technique may lead to significative reduction of connection penalties due to long round trip times.

Let’s now focus about the way the algorithm works. One process, the ”queuer”, communicates with the netfilter by the ipqueue library routines. It reads every syn packet in the queue, annotating the information on the two endpoints in a known zone of the shared memory and then release the packet which continues its path through the netfilter chain. Just after that, the SYN packet is redirected by netfilter to tcp port 5000, where a TCP daemon, the ”connection manager” is listening for it. The connection is accepted by a dedicated ”proxy server” process, which searches in the shared memory for the instance matching source address and TCP port of the host which has started the connection. Once the destination IP address and port have been found in the connection array, a new TCP connection is attempted to real destination. When both connections are established, the proxy spawns two little concurrent threads, each one reading from one TCP socket and writing all the data to the other one. When one of the two connections ends, the other socket is closed and the proxy process ends.

In the end we can say that PEPsal represents a valid solution for the degraded TCP performances when satellite link are involved. Its approach can surely be of great inspiration for the future developments of PEP algorithms.

### 3.2.3 TCPeP

The algorithm that has been used as first is the TCPeP. We are going to it as the last one as it is the one that offers the most important results in terms of performance improvements. This name derives from the Transport Control Protocol and Performance-Enhancing Proxy achronymouses. This gives us the information that this algorithm is used for improving the performances of services based on the TCP protocol, and is thought to be deployed by two proxies that communicate over lossy links. It is based on Network Coding principles and the previously analyzed ”Network coded tcp - CTCP” protocol scheme, also in this case implemented in combination with the TCP splitted connection approach. This solution comes to be useful as the standard TCP algorithms were designed with wired networks in mind, thus assuming that all losses resulted from congestion events. Coded-TCP, instead, is based on the assumption that losses can be completely random, as it occurs in wireless links.

Finally, it’s necessary to specify that even though the implemented version of the CTCP protocol is a complete Transport-Layer protocol, it is not practically possible to encapsulate it directly on top of the Network-Layer protocol, IP, as most firewalls and routers can only recognize- and
CHAPTER 3. STATE OF THE ART OF THE USED TECHNOLOGIES

Figure 3.9: Sequence diagram summarizing the PEPsal algorithm behaviour
therefore transmit- standardized TCP or UDP packets. Therefore, as we will see in the captures shown in the next sections, the communication will so take place on top of UDP, with one duplex UDP flow per Client.

**Setting up the scenario**

Now we try to give a quick description about how the algorithm works in an explicative scenario made of three hosts. The scenario in shown in Figure 3.10.

This scenario is only composed of three hosts. This is because the algorithm may also be directly run in the endhost. In this way the host itself also plays the role of the PEP. According to the guide that can be found in the Github website, to let the algorithm work it’s only necessary to download the source, compile it, add some modifications to the source PEP iptables and, finally, execute the tcpep.sh script as shown below. The command used to get the source folder is

```
git clone "https://github.com/GregoireDelannoy/TCPeP.git"
```

Then, once compiled the whole content of the downloaded folder, we need to add some rules to the iptables of the source-side proxy (that can also be the source host itself). The only difference between the case in which we run the proxy in the sender endhost itself and the one in which we run it in a separate host stands in the chain where we add the new rule, i.e. the OUTPUT chain in the former case, the PREROUTING chain in the last. Let’s give a look to what these rules are:

```
iptables -t nat -A OUTPUT|PREROUTING -p tcp -d <dest_host_IP> -j REDIRECT --to-ports <xxxx>
```

This command adds to the NAT table a rule that redirects to a certain port all of the TCP packets to be forwarded towards the specified IP address. The specified port is the one where the TCPeP

![Figure 3.10: Scenario map created using scenario2.xml](image-url)
process will be indicated to listen to

iptables -t nat -A OUTPUT -p tcp ! -d 127.0.0.1 -j REDIRECT --to-ports <xxxx>

This command performs the same operation of the former to all of the tcp packets not directed to the local host. It only makes sense when the TCPeP client proxy is run in the localhost. Then we type, respectively, in the source and destination proxies the following commands:

./tcpep -C <remote_proxy_IP> -u yyyy -t xxxx
./tcpep -P -u yyyy

In the source proxy we have the TCPeP proxy listening to the port "xxxx" and redirecting the tcp packets received through this port to the port "yyyy" of the remote proxy after having converted them in UDP packet, thus establishing a UDP connection between the two proxies. Finally, it’s important to specify that at the end of this test we need to remove these rules from the iptables if we want to test other algorithms. To do it, we just need to execute the following command:

iptables -t nat -F

Analyzing the traffic

In order to better explain the way the algorithm works, we will analyze the traffic both when the physical host behaves as a client and the remote host as a server and viceversa. Obviously, in this case we want to see how the algorithm treats the TCP packets that client and server want to exchange. For this reason, the ICMP ping is no longer a satisfying test. The easiest way to go is using the netcat tool, which allows the exchange of TCP packets.

We start trying to set up a message exchange from the physical host to the remote host. In this case, the physical host also behaves as the source PEP, while the PEP behaves as destination PEP. We subdivide the scenario in three branches:

- in the first one we can see the packets captured by the loopback interface, that are the packets redirected from the exit port to the port the TCPeP process is listening to (Figure 3.11);

- in the second branch we can see the packets exchanged between the TCPeP process in the physical host and the PEP. In the capture shown in Figure 3.12, we can see that the two TCPeP processes continuously exchange UDP packets, and that also the content of the messages exchanged through the netcat application are here carried by UDP packets, thus giving a proof about the fact that the algorithm is correctly working;

- finally, in the third branch we can see the packets exchanged between the PEP and the remote host (Figure 3.13). Here we notice that a new TCP connection comes to be established, and
### Figure 3.11: Packets exchanged through the loopback interface

<table>
<thead>
<tr>
<th>No.</th>
<th>Time</th>
<th>Source</th>
<th>Destination</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>11416</td>
<td>20:55:32,32672034</td>
<td>192.168.1.1</td>
<td>127.0.0.1</td>
<td>TCP</td>
</tr>
<tr>
<td>11417</td>
<td>20:55:32,32736783</td>
<td>192.168.2.2</td>
<td>192.168.1.1</td>
<td>TCP</td>
</tr>
<tr>
<td>11418</td>
<td>20:55:32,32978259</td>
<td>192.168.1.1</td>
<td>127.0.0.1</td>
<td>TCP</td>
</tr>
<tr>
<td>11693</td>
<td>20:55:35,613135829</td>
<td>192.168.1.1</td>
<td>127.0.0.1</td>
<td>TCP</td>
</tr>
<tr>
<td>11694</td>
<td>20:55:35,612215746</td>
<td>192.168.2.2</td>
<td>192.168.1.1</td>
<td>TCP</td>
</tr>
<tr>
<td>11913</td>
<td>20:55:39,605457100</td>
<td>192.168.2.2</td>
<td>192.168.1.1</td>
<td>TCP</td>
</tr>
<tr>
<td>11194</td>
<td>20:55:39,604596884</td>
<td>192.168.1.1</td>
<td>127.0.0.1</td>
<td>TCP</td>
</tr>
</tbody>
</table>

### Figure 3.12: Packets exchanged through the feeder interface

<table>
<thead>
<tr>
<th>No.</th>
<th>Time</th>
<th>Source</th>
<th>Destination</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20:55:32,321424586</td>
<td>192.168.1.1</td>
<td>192.168.1.2</td>
<td>UDP</td>
</tr>
<tr>
<td>2</td>
<td>20:55:33,415244542</td>
<td>192.168.1.1</td>
<td>192.168.1.2</td>
<td>UDP</td>
</tr>
<tr>
<td>3</td>
<td>20:55:33,791405111</td>
<td>192.168.1.1</td>
<td>192.168.1.2</td>
<td>UDP</td>
</tr>
<tr>
<td>4</td>
<td>20:55:34,792893363</td>
<td>192.168.1.1</td>
<td>192.168.1.2</td>
<td>UDP</td>
</tr>
<tr>
<td>5</td>
<td>20:55:35,613542789</td>
<td>192.168.1.2</td>
<td>192.168.1.1</td>
<td>UDP</td>
</tr>
<tr>
<td>6</td>
<td>20:55:35,613542789</td>
<td>192.168.1.2</td>
<td>192.168.1.1</td>
<td>UDP</td>
</tr>
<tr>
<td>7</td>
<td>20:55:39,603784383</td>
<td>192.168.1.1</td>
<td>192.168.1.2</td>
<td>UDP</td>
</tr>
<tr>
<td>8</td>
<td>20:55:39,603844418</td>
<td>192.168.1.2</td>
<td>192.168.1.1</td>
<td>UDP</td>
</tr>
<tr>
<td>9</td>
<td>20:55:39,604250332</td>
<td>192.168.1.2</td>
<td>192.168.1.1</td>
<td>UDP</td>
</tr>
</tbody>
</table>

### Figure 3.13: Packets exchanged through the sinker interface

<table>
<thead>
<tr>
<th>No.</th>
<th>Time</th>
<th>Source</th>
<th>Destination</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20:55:32,32547235</td>
<td>192.168.2.1</td>
<td>192.168.2.2</td>
<td>TCP</td>
</tr>
<tr>
<td>2</td>
<td>20:55:32,323003445</td>
<td>192.168.2.2</td>
<td>192.168.2.1</td>
<td>TCP</td>
</tr>
<tr>
<td>3</td>
<td>20:55:37,32380990</td>
<td>192.168.2.1</td>
<td>192.168.2.2</td>
<td>TCP</td>
</tr>
<tr>
<td>4</td>
<td>20:55:35,61340395</td>
<td>192.168.2.2</td>
<td>192.168.2.1</td>
<td>TCP</td>
</tr>
<tr>
<td>5</td>
<td>20:55:39,604290809</td>
<td>192.168.2.1</td>
<td>192.168.2.2</td>
<td>TCP</td>
</tr>
<tr>
<td>6</td>
<td>20:55:39,604522866</td>
<td>192.168.2.2</td>
<td>192.168.2.1</td>
<td>TCP</td>
</tr>
<tr>
<td>7</td>
<td>20:55:39,604601831</td>
<td>192.168.2.1</td>
<td>192.168.2.2</td>
<td>TCP</td>
</tr>
</tbody>
</table>
the UDP packets containing the exchanged messages received by the PEP are converted again in TCP packets.

Let’s give a deeper look to the loopback interface capture: according to the theoretical behaviour of the algorithm, we should have a TCP communications between the PEP processes and their relative end nodes, while the packets exchanged between the two PEP themselves should travel over UDP. By the way, what we can see from this capture is that the TCP connection between the remote host and the PEP is established and detached as expected, while in the other TCP connection, the one that should take place between the physical host and its own loopback interface, the response messages appear to be sent by the remote node. What just described doesn’t make sense, as the remote node shouldn’t be aware of the messages that the local host sends to its own loopback interface. Besides, if this message exchange should really take place between the two end nodes, it would mean that the algorithm would be really inefficient to be deployed in environments with significative the end-to-end delays. So, in order to better get acquainted about what’s actually happening, we will try to give a better explanation to this in the next sections, when we will to analyze the behaviour of this algorithm in a scenario that simulates a satellite network.
Chapter 4

Experimental setup

In this section it will be explained how the experimental tests of those algorithms have been pur-sued: in the first section the simulated topology is explained; then the way this topology has been analyzed through tests is described; finally, the results collected for each one of the tested algo-

rithms are compared.

As stated in the previous sections, the main feature that characterizes this work is that the envi-

ronment that is going to be studied is a satellite network, which has different features with respect to ethernet-wired networks. The scenario over which the tests have been performed has been set up according to the Split-Connection approach.

GEO satellite networks are characterized by a long two-way propagation delay, that results in a long minimum Round Trip Time (RTT). This causes TCP connections to take up to several seconds before reacting to or recovering from congestions. This architecture has been proposed to optimize the transport layer performance in satellite environments, in the particular context of the new generation DVB-S2 standard for Interactive Services. GEO satellite communications offer the advantages of allowing to provide communication services at anytime and everywhere and being suitable for scenarios where it is difficult or impossible to deploy wired communication infrastructures (e.g. in rural environments, developing countries or airplane/boat communications). On the other hand, we cannot neglect som drawbacks as: variable link capacity, high propagation delays, packet corruption and channel asymmetry. For this reason, applications that use the reliable transport protocol TCP (Transmission Control Protocol) have a significant degradation over satellite links. This is mainly because of the congestion control algorithm of standard TCP. A variety of solutions in which network devices send feedback to the TCP layer to enhance its performance have been proposed to address the mentioned problems. This approach requires satellite-optimized TCP protocols to run on an end-to-end user basis. However, forcing all the users (including Internet servers) to run a satellite-optimized TCP is not achievable in practice.
Moreover, in this case end users should be able to foresee if the path includes a satellite link or not to apply the proper TCP-flavor. To mitigate the mentioned problems, the most promising solution for satellite networks is splitting, which consists of dividing the TCP connection into a satellite portion and a terrestrial portion. The objective of splitting is isolating the long-latency satellite link with middle agents called Performance-Enhancing Proxies (PEPs). With splitting, the connection is divided in three TCP connections: sender-to-PEP, PEP-to-PEP and PEP-to-receiver. PEPs are responsible for intercepting, caching, acknowledge data received by senders and forwarding these data to receivers. In this scheme there are two elements that manage the physical medium and the data link layer of the satellite link: a satellite forwarder node (HUB) and a Return Channel Satellite Terminal (RCST). Each satellite subscriber has a RCST, while the satellite operator manages the HUB. The link HUB-to-RCST is called forward link and it is defined by the DVB-S2 standard. The RCST-to-HUB link is called return channel and it is defined by the DVB-RCS standard. In practice, the HUB is located in the infrastructure side (which belongs to the operator) where it is also connected to the Internet. Therefore, the forward link is intensively used by satellite terminals in their access to Internet applications and services. The overall scenario is reported in Figure 4.1.

4.1 Building the scenario

Also in this case, the scenario has been developed using the VNX software. The realized virtual architecture is composed of six elements, thus forming a symmetric topology, where:

- the physical host and the remote host container represent the end-to-end nodes of the link;
- the satpep and pep-m nodes implement the algorithms of the different types of PEPs implemented;
CHAPTER 4. EXPERIMENTAL SETUP

The central nodes, saterm and hubm, emulate the conditions that a packet would experience crossing a satellite network, thus representing the border devices of the satellite link.

The idea of the topology is to deploy a proxy for each of the end host in a way to address the issues that may occur over the lossy link. This idea is schematized in Figure 4.1.

The realized scenario is shown in Figure 4.2, while Figure 4.3 gives a complete representation about how the scenario is organized inside the physical host kernel.
CHAPTER 4. EXPERIMENTAL SETUP

4.2 Performance evaluations

4.2.1 Latency test

The latency test has been performed as first in order to check the end-to-end delay experienced by a packet that crosses the whole scenario. As the machines we are using lay in the same physical node, the original latency is very small, so we need to use the Linux traffic control software in a way to reproduce the latency conditions of a satellite network. In case of geostationary satellite networks a good value to be considered is 285 ms (i.e. 570ms of RTT).

Before to check how we have set the system up and the results of these settings, it’s important to bare in mind that the traffic control is a tool that basically allows the setup of the queuing disciplines of our VMs interfaces, i.e. the rules that those interface must apply when packets are forwarded through it.

As regards the aspect concerning the latency features of the scenario, we have used the Network Emulator (netem) discipline, which allows simulating the properties of wide area networks. Let’s quickly analyze the syntax of the used commands:

- **tc**, allows to manipulate or show the traffic control settings;

- **qdisc dev eth2**, indicates that we are working on queuing disciplines of the eth2 interface;

- **root**, means that the discipline is attached to the root of the interface;

- **handle 1:0**, is a unique identifier of the discipline, only needed in case of classful qdiscs;

- **estimator 250ms 361ms**, allows to perform an estimation of the bandwidth used by each class every 250 ms through an Exponential Weighted Moving Average (EWMA). The 361 ms timeconstant indicates the sensitivity of the computed average to short bursts;

- **netem**, it’s the name of the used queuing discipline;

- **delay 285ms**, delays the forwarding operation of the indicated amount of time.

- **netem**, limits the maximum number of packets that can be on flight without being acknowledged.

To obtain the desired feature, we need to set the netem as follows:

- on the interface of the remote terminal of the satellite link that is external with respect to the link itself (eth2 in the hubm VM for the UL and eth1 in the satterm VM for the DL), in this way (DL example)

  ```bash
  root@satterm:~ # tc qdisc replace dev eth1 root handle 1:0 estimator 250ms 361ms
  netem delay 285ms limit 3000 loss random 1%
  ```
In this way we are both simulating a $10^{-2}$ packet error rate, and a 285 ms delay in the forward path.

- on the interface of the local terminal of the satellite link that is external with respect to the link itself (eth2 in the hub for the DL and eth1 in the satterm for the UL), in this way (DL example)

```bash
root@hub:~# tc qdisc replace dev eth2 root handle 1:0 estimator 250ms 361ms netem delay 285ms limit 3000
```

In this way we simulate a 285ms end-to-end delay, thus obtaining an overall latency (RTT) of 570ms, as shown in the results below.

```bash
root@remote:~# ping 192.168.200.1
PING 192.168.200.1 (192.168.200.1) 56(84) bytes of data.
64 bytes from 192.168.200.1: icmp_seq=1 ttl=60 time=570 ms
64 bytes from 192.168.200.1: icmp_seq=2 ttl=60 time=570 ms
64 bytes from 192.168.200.1: icmp_seq=3 ttl=60 time=570 ms
```

Before performing the next tests, we also need to set the maximum throughput at which we want our senders to transmit packets throughout the scenario. To do this, it is possible to use the Token Bucket Filter (tbf) discipline, which is based on the Token Bucket. This algorithm virtually creates a bucket in which it keeps putting tokens at a certain rate. A packet in the queue can be forwarded by the interface that applies the algorithm only if the bucket contains a token, otherwise it remains in the queue, waiting for a new token to be generated. Moreover, the bucket size limits the number of instantaneously available tokens (Figure 4.4). In this way, the rate at which tokens are generated shapes the traffic, setting a limit to the maximum achievable throughput. Delaying packets in this way will also introduce an artificial latency into the packet’s round trip time. The syntax of the tbf discipline is made of the following elements:

- **tbf**, is the name of the used discipline;
- **rate**, sets the maximum achievable bitrate;
- **burst 6120**, sets the buffer size;
- **lat 570ms**, specifies the maximum amount of time a packet can wait in the TBF;
- **mtu 1540**, specifies the size of the peakrate bucket. For perfect accuracy, it should be set to the MTU of the interface.
CHAPTER 4. EXPERIMENTAL SETUP

This discipline has to be set on the interface of the local terminal of the satellite link that is internal with respect to the link itself (eth2 in the hub for the DL and eth1 in the satterm for the UL), as shown below (DL example):

```
root@hub:~# tc qdisc replace dev eth1 root estimator 250ms 361ms tbf rate 18000kbit burst 6120 lat 570ms mtu 1540
```

Finally, in figure 4.5 it is showed how the interfaces of the saterm and hubm VMs have to be set in order to have a link with 570 ms of round trip time, a 2 MB peakrate and $10^{-2}$ packet error rate in uplink (from local to remote) and 18 MB and $10^{-4}$ in downlink.

After having described how the virtual satellite link has been set, we can now go on and see the results of the performed tests.
4.2.2 Throughput test

In this section, the results of the performed tests are presented. As the goal of this work is to establish whether the benefits introduced by the studied performance enhancing algorithms are consistent or not, we have tested as well the scenario in the situation where none of these algorithms has been used. The simulations have been conducted in a way to test the scenario under different packet error probabilities. For each error probability value, the following types of transfers have been performed, in a way to enable us to observe in which of those cases the used algorithms perform better:

- a short transfer has been executed by sending a 10 kilobyte file;
- a medium transfer has been executed by sending a 15 Megabyte file;
- a long transfer has been executed by sending a 90 Megabyte file.

End-to-end all TCP Reno scenario

As stated in the previous paragraph, we start testing the scenario without implementing any performance algorithm in the nodes that are supposed to act as PEPs. In this way, those nodes simply act as border routers that allow the transition of packets from the local (or remote) host LAN to the opposite end, passing through the virtual satellite link.

The following captures show the traffic flow through the different branches of the network when a simple netcat conversation is established and concluded. In the screenshot of Figure 4.6 it is possible to notice that a 570 milliseconds time interval intercurs between the moment in which the client sends the SYN packet and the one in which it receives the SYN+ACK packet by the receiver. This is due to the fact that in this case the two TCP entities that interact in this exchange are the local and the remote host, without any intermediate split at transport layer.

For this reason, the client is forced to wait for 570 milliseconds before being able to send messages to be used by the server at application layer. Obviously, the aforementioned captures also allow to check that this delay is both consistent in all of the observed branches and for every packet exchange. So, the behaviour of the TCP protocol over this environment can be summarized as shown by the sequence diagram in Figure 4.9.

After having described in details how the whole system works, we can now describe how the algorithm has been tested and expose the results. The tests results are exposed by reporting the I/O charts obtained using the IOgraphs tool of the wireshark software. Only in the cases of the short transfers, we will represent data by means of Stevens diagrams\(^1\). We start analyzing the short transfer case in the Uplink direction.

\(^1\)The x-axis is time. The y-axis is TCP sequence numbers. Sequence numbers are representative of bytes sent.
CHAPTER 4. EXPERIMENTAL SETUP

55

Figure 4.6: Packets captured by the feeder device

Figure 4.7: Packets captured by the satellite device

Figure 4.8: Packets captured by the remote device
As shown in Figure 4.10, in this case the diagram assumes the same behaviour for each of the considered error probability values. More specifically we notice that in all of those cases, no losses occur. For this reason, the cwnd can be doubled every round trip time (570 milliseconds), and the transfer finishes before that a congestion event may occur, as for the slow-start mechanism of the TCP Reno congestion control algorithm.

Things start behaving differently when we experiment longer transfers. As shown in figure 4.11, when we use the classic TCP Reno approach, the cwnd value first reaches, in a very few time, a peak value that causes congestion. After this first congestion event, the window is taken decreased and starts the congestion avoidance phase. As we can see, its time behaviour varies a lot according to the amount of losses: when the packet loss probability is high, the window is not able at all to assume to its maximum value; with medium packet loss probability value, the window is hardly able to get to its maximum value; only in case of low packet loss probability, the window appears

The sequence number increases by 1 for every 1 byte of TCP data sent. Ideally you’d want to see a smooth line going up and to the right. The slope of the line would be the theoretical bandwidth of the pipe. The steeper the line, the higher the throughput. The little black I-beams represent TCP data segments. The longer the I-beam, the more data per packet.
to be able to reach its maximum value and assume it constantly.

As confirmed by the diagrams in Figure 4.12, this behaviour is totally coherent with what happens in the long transfer case. Also in this case, in fact, we can see that: in the case of one percent packet loss rate the throughput not even remotely reaches its nominal value (2 Mbits per second for the UL case), as the Reno congestion avoidance mechanism result to be too slow; in the medium packet loss rate case it almost reaches this value, but is definitely unable to hold it constantly; in the low loss probability case it finally succeeds doing it. By the way, also in this case the result is not so good, as the congestion avoidance mechanism results to need around twenty seconds without losses to guarantee a constant rate, which is such a big requirement.

Finally, it is also easy to notice that while the short transfers times are the same for every packet error rate value, these times vary a lot in the longer transfer cases. This is what results to happen as a consequence of the fact that in the short transfers the this time only depends on the slow-start algorithm adopted, as no congestion events result to be detected.
Figure 4.11: Throughput curves obtained by performing the medium transfer in the Uplink direction with a packet error probability amount of respectively $10^{-2}$, $10^{-3}$ and $10^{-4}$
Figure 4.12: Throughput curves obtained by performing the long transfer in the Uplink direction with a packet error probability amount of respectively $10^{-2}$, $10^{-3}$ and $10^{-4}$
CHAPTER 4. EXPERIMENTAL SETUP

Figure 4.13: Stevens diagrams obtained by performing a short transfer in the Downlink direction with a packet error probability amount of $10^{-2}$, $10^{-3}$ and $10^{-4}$.

Figures 4.13, 4.14 and 4.15 show what happens in the same cases in the Downlink direction. More specifically, it can be observed that the short transfer behaves exactly as it does in the Uplink case as the TCP mechanisms are symmetric in both of the directions, and the client doesn’t detect any congestion. For these reasons the transfer is all included during the slow-start phase. Moreover, by observing the first part of the longer transfers diagrams (Figures 4.14 and 4.15), in the Downlink case it’s easier to notice the exponential behaviour of the throughput during this phase: in the Uplink case, instead, it was not so easy to be observed as the throughput reached very fastly the peak value (even because it was of only two MBits per second instead of the almost eighteen of the Downlink case), so the time interval in which this exponential characteristic got developed was very neglectable with respect to the whole transfer one.
By observing the rest of the curve in medium transfer case, it is possible to observe that:

- once the nominal throughput value is reached during the slow start, the algorithm ends this phase and keeps it constant as much as possible (i.e. until a loss occurs). For this reason the throughput behaviour is almost the same both for the low and the medium loss probability values, as the remote sender is able to transfer the whole file during this phase.

- high packet loss probability dramatically affect the throughput performance with respect to the other cases, in fact forcing the throughput to a very low value with respect to the long and medium transfer cases. This situation remarks the fact that classic TCP performances over lossy links are very poor, so it’s necessary to adopt alternative solutions.

So, while the first observation may suggest us that the TCP behaviour over lossy links may not be such a critical problem, the second one, instead, tells us right the opposite thing, as an one order of magnitude increase is enough to totally compromise the situation.

Things appear more and more dramatic if we analyze the long transfer plots (Figure 4.15): here we observe that once a loss occurs the algorithm struggles a lot to reach again the nominal value, so the performances result to be acceptable only in the low loss probability case.

At the end of this section, it’s possible to summarize what has been observed saying that the performed simulations have demonstrated what we had anticipated in the previous chapter, when we have talked about the TCP issues when it operates over lossy links with long latencies. In the next sections we will evaluate how well do perform some of the solutions that have been studied and designed so far.
Figure 4.14: Throughput curves obtained by performing the medium transfer in the Downlink direction with a packet error probability amount of respectively $10^{-2}$, $10^{-3}$ and $10^{-4}$.
Figure 4.15: Throughput curves obtained by performing the long transfer in the Downlink direction with a packet error probability amount of respectively $10^{-2}$, $10^{-3}$ and $10^{-4}$
CHAPTER 4. EXPERIMENTAL SETUP

PEPsal scenario

According to the considerations done in the previous paragraph, we now try to see if it is possible to obtain benefits by implementing two of the PEP algorithms that have been analyzed in the third chapter. We start discussing about the PEPsal algorithm.

As described in chapter three, the benefits that this algorithm may contribute to obtain lay on the use of a TCP version based on a more aggressive congestion control algorithm, rather than on the use of FEC techniques.

We have run the PEPsal algorithm in our test scenario by simply invoking it through the following codelines of the scenario xml configuration file.

```
<filetree seq="on_boot" root="/tmp/">files/satpep/progs</filetree>
<filetree seq="on_boot" root="/etc/pepsal/">files/satpep/etc/</filetree>
<filetree seq="pep_cfg" root="/tmp/progs/">files/satpep/progs/pep.cfg</filetree>
<exec seq="on_boot" type="file" ostype="system"> files/satpep/setup/fq.sh</exec>
<exec seq="on_boot" type="file" ostype="system"> files/satpep/setup/ip.sh</exec>
<exec seq="on_boot" type="verbatim" ostype="system"> /tmp/progs/set_mangle.sh eth2</exec>
<exec seq="pep_start" type="verbatim" ostype="exec">nohup /tmp/progs/pep.sh start</exec>
<exec seq="pep_stop" type="verbatim" ostype="exec">nohup /tmp/progs/pep.sh stop</exec>
```

These lines copy the source files of the PEPsal from the local host kernel to the kernels of the PEP containers, and associate the execution of them in a way to configure the PEPs thus allowing them intercepting the TCP packets and performing the layering and splitting functions.

The commands to be executed in the local host root terminal in order to configure, start or stop the PEP in those machines are expressed by the following line:

```
vnx -f /path/to/the/vnx/file.xml --execute pep_cfg|pep_start|pep_stop
```

Moreover, it is also possible to freely decide which congestion avoidance algorithm to deploy by writing it in the `pep.cfg` file. In our case the deployed algorithm has been the Cubic [11], which is an algorithm designed for high-speed network environments as an enhanced version of the BIC [12] algorithm that simplifies its window control and improves its TCP-friendliness\(^2\) and RTT-fairness\(^3\).

\(^2\)The ability of a system to behave under congestion like the TCP protocol.
\(^3\)Ratio of the throughput achieved by two groups of flows as a function of the ratio of their RTTs.
In the previous paragraph it has been observed that the short transfers behaviour is very useful to be observed, as it gives us very important informations about the way the congestion control mechanism adopted by the studied algorithm behaves. By the way, having already shown that its behaviour is completely symmetric in both the link directions, it’s enough to observe how it works in only one of them (e.g. the Uplink). By simply observing the time axis of this graph and comparing it with the one obtained in the previous case (end-to-end TCP Reno) it can easily be noticed how the algorithm implemented by the PEPsal is way faster than the previous one. The effects of this peculiarity of the PEPsal approach are more visible in the longer transfers. In fact, in the graphs in Figure 4.18 and 4.19 show that, in the Uplink direction, even if the provided throughput is not constant at all, the algorithm allows a very quick recovery, thus providing an acceptable average throughput. The reason why the throughput drops so badly when losses occur is to be associated to the fact that the PEPsal just allow splitting the connection and using a faster approach where it is more necessary, that is over the lossy link. On the other hand, the weakness of this algorithm lays on the fact that the protocol used over the lossy link is still the TCP, so it’s affected by of all of the issues we have been talinkg so far.

Things go way better in the Downlink direction, thus indicating that this algorithm is fast enough in speedth recovery to almost succeed exploiting the whole channel capacity when the packet error rate is lower.
Figure 4.17: Throughput curves obtained by performing the medium transfer in the Uplink direction with a packet error probability amount of respectively $10^{-2}$, $10^{-3}$ and $10^{-4}$
Figure 4.18: Throughput curves obtained by performing the long transfer in the Uplink direction with a packet error probability amount of respectively $10^{-2}$, $10^{-3}$ and $10^{-4}$.
Figure 4.19: Throughput curves obtained by performing the medium transfer in the Downlink direction with a packet error probability amount of respectively $10^{-2}$, $10^{-3}$ and $10^{-4}$
Figure 4.20: Throughput curves obtained by performing the long transfer in the Downlink direction with a packet error probability amount of respectively $10^{-2}$, $10^{-3}$ and $10^{-4}$. 
At the end of this section, we can conclude that the results obtained employing this solution are very interesting: as a result of the carried out analysis we can say that the implemented algorithm has surely improved the overall performance. More specifically these results mainly show off that:

- in case of environments that are only barely lossy (i.e. $P_e = 10^{-4}$), the use of highly aggressive congestion control algorithms allow the system to exploit almost all of the available bandwidth;

- the fact that its performances drop down when losses become relevant suggests that the PEPsal approach can be intended to be used when thinking about designing high-performance PEP algorithms in parallel to other techniques that allow addressing the problems related to the use of a connection-oriented protocol (e.g. avoiding retransmission mechanisms, using FECs, and so on)
TCPeP scenario

The other algorithm that we have decided to test is the TCPeP. As described in the previous chapter, the TCPeP approach is based on the adoption of the FEC technique. On the other hand, this algorithm doesn’t upset the TCP friendliness concept, as it is based on a transport protocol known as Coded TCP which basically introduces the FEC functions in order to improve the performances over long lossy links, but still maintaining the intent of having a reliable end-to-end connection.

As for the PEPsal, also the TCPeP is based on the deployment of layering and splitting functions. The configuration of the scenario is done as follows.

In the local PEP virtual machine, the following commands have to be executed:

```
iptables -t nat -A PREROUTING -p tcp -d IP_dest_host -j REDIRECT --to-ports tcpep
-application_port
```

In particular, the `iptables` editing command shows off the layering concept: when the local proxy receives an incoming packet from the local end-host, first of all it analyzes it at network layer. If its destination IP address is not the same of the local proxy itself and coincides with the one indicated in the iptables, then it sends it to the upper layer iptable, otherwise it forwards it according to its routing tables. If a packet that arrives to the transport layer iptable is a TCP packet and has a destination port that coincides with the one specified in the aforementioned command, then the packet is sent to the process at the application layer that is listening to that port. The second command, instead, launches the `tcpep` application process, indicating it to intercept packets arriving from the specified TCP (`-t`) port. This port needs to be the same port specified in the iptables command. Once the application processes the packet, then the new application data is encapsulated in a UDP header with a new destination port (specified with the `−u` option), and then forwarded to the original destination, according to the proxy routing tables.

In the remote PEP, the following command is executed:

```
./tcpep -C IP_remotePEP -t tcpep-application_port -u remotePEP_listening-port
```

In particular, the `iptables` editing command shows off the layering concept: when the local proxy receives an incoming packet from the local end-host, first of all it analyzes it at network layer. If its destination IP address is not the same of the local proxy itself and coincides with the one indicated in the iptables, then it sends it to the upper layer iptable, otherwise it forwards it according to its routing tables. If a packet that arrives to the transport layer iptable is a TCP packet and has a destination port that coincides with the one specified in the aforementioned command, then the packet is sent to the process at the application layer that is listening to that port. The second command, instead, launches the `tcpep` application process, indicating it to intercept packets arriving from the specified TCP (`-t`) port. This port needs to be the same port specified in the iptables command. Once the application processes the packet, then the new application data is encapsulated in a UDP header with a new destination port (specified with the `−u` option), and then forwarded to the original destination, according to the proxy routing tables.

In the remote PEP, the following command is executed:

```
./tcpep -P -u remotePEP_listening-port
```

It indicates that in this machine, the tcpep application process is listening to the same UDP port as the one specified at the local proxy side. Once arrived there, packets are processed and forwarded to the final user, encapsulated again in a TCP header.
CHAPTER 4. EXPERIMENTAL SETUP

72

<table>
<thead>
<tr>
<th>No.</th>
<th>Time</th>
<th>Source</th>
<th>Destination</th>
<th>Protocol</th>
<th>Length</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0000000000</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>74</td>
<td>55492 - 1234 [SYN] Seq=6</td>
</tr>
<tr>
<td>2</td>
<td>0.0000000000</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>74</td>
<td>55492 - 1234 [ACK] Ack=7</td>
</tr>
<tr>
<td>3</td>
<td>0.0000000000</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>66</td>
<td>55492 - 1234 [ACK] Ack=4</td>
</tr>
<tr>
<td>5</td>
<td>5.2.4349528156</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>66</td>
<td>55492 - 1234 [ACK] Ack=4</td>
</tr>
<tr>
<td>7</td>
<td>5.2.4349528156</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>66</td>
<td>55492 - 1234 [ACK] Ack=4</td>
</tr>
<tr>
<td>8</td>
<td>5.2.4349528156</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>66</td>
<td>55492 - 1234 [ACK] Ack=4</td>
</tr>
</tbody>
</table>

Frame 4: 71 bytes on wire (568 bits), 71 bytes captured (568 bits) on interface 0
- Ethernet II, Src: 00:50:56:74:9d:69 (02:00:00:00:00:00), Dst: 02:ff:ee:00:00:00 (00:00:00:00:00:00)
- Transmission Control Protocol, Src Port: 55492, Dst Port: 1234, Seq: 1, Ack: 1, Len: 5

Figure 4.21: Packets captured by the remotebr device

<table>
<thead>
<tr>
<th>No.</th>
<th>Time</th>
<th>Source</th>
<th>Destination</th>
<th>Protocol</th>
<th>Length</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.2.4349528156</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>53</td>
<td>43638 - 2006 Len=11</td>
</tr>
<tr>
<td>2</td>
<td>5.2.4349528156</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>53</td>
<td>43638 - 2006 Len=11</td>
</tr>
<tr>
<td>3</td>
<td>5.2.4349528156</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>53</td>
<td>43638 - 2006 Len=11</td>
</tr>
<tr>
<td>4</td>
<td>5.2.4349528156</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>53</td>
<td>43638 - 2006 Len=11</td>
</tr>
<tr>
<td>5</td>
<td>5.2.4349528156</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>53</td>
<td>43638 - 2006 Len=11</td>
</tr>
<tr>
<td>6</td>
<td>5.2.4349528156</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>53</td>
<td>43638 - 2006 Len=11</td>
</tr>
<tr>
<td>7</td>
<td>5.2.4349528156</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>53</td>
<td>43638 - 2006 Len=11</td>
</tr>
<tr>
<td>8</td>
<td>5.2.4349528156</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>53</td>
<td>43638 - 2006 Len=11</td>
</tr>
<tr>
<td>9</td>
<td>5.2.4349528156</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>53</td>
<td>43638 - 2006 Len=11</td>
</tr>
</tbody>
</table>

Ethernet II, Src: 02:ff:ee:00:00:00 (00:00:00:00:00:00), Dst: 02:ff:ee:00:00:00 (00:00:00:00:00:00)

Figure 4.22: Packets captured by the remotebr device

<table>
<thead>
<tr>
<th>No.</th>
<th>Time</th>
<th>Source</th>
<th>Destination</th>
<th>Protocol</th>
<th>Length</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0000000000</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>53</td>
<td>43638 - 2006 Len=11</td>
</tr>
<tr>
<td>2</td>
<td>0.0000000000</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>53</td>
<td>43638 - 2006 Len=11</td>
</tr>
<tr>
<td>3</td>
<td>0.0000000000</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>53</td>
<td>43638 - 2006 Len=11</td>
</tr>
<tr>
<td>4</td>
<td>0.0000000000</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>53</td>
<td>43638 - 2006 Len=11</td>
</tr>
<tr>
<td>5</td>
<td>0.0000000000</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>53</td>
<td>43638 - 2006 Len=11</td>
</tr>
<tr>
<td>6</td>
<td>0.0000000000</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>53</td>
<td>43638 - 2006 Len=11</td>
</tr>
<tr>
<td>7</td>
<td>0.0000000000</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>53</td>
<td>43638 - 2006 Len=11</td>
</tr>
<tr>
<td>8</td>
<td>0.0000000000</td>
<td>192.168.200.1</td>
<td>192.168.200.1</td>
<td>TCP</td>
<td>53</td>
<td>43638 - 2006 Len=11</td>
</tr>
</tbody>
</table>

Frame 4: 71 bytes on wire (568 bits), 71 bytes captured (568 bits) on interface 0
- Ethernet II, Src: 02:ff:ee:00:00:00 (00:00:00:00:00:00), Dst: 02:ff:ee:00:00:00 (00:00:00:00:00:00)
- Transmission Control Protocol, Src Port: 43638, Dst Port: 1234, Seq: 1, Ack: 1, Len: 5

Figure 4.23: Packets captured by the remotebr device
The described procedure also highlights the connection split in terms of transport layer protocol between the remote LANs and the central satellite lossy link, which is also very well shown both in the captures of Figure 4.23 and in the sequence diagram in Figure 4.24.

In both of these pictures, it’s immediately possible to notice that the split connection approach allows making the whole system faster removing the time intervals in which each endhost should have waited for the ack to come from the other side of the network. The price of this trick is that the connection stops to be reliable. The FEC technique is the tool that this algorithm use to try to make the system anyway reliable.

Analyzing the results shown in the next pictures, it is possible to observe that:

- in the short transfer (Figure 4.25), the algorithm seem to perform worse with respect to both of the previously analyzed cases. In particular, we notice that the window mantains a constant width in time, thus resulting way slower. This happens as a consequence of the congestion control algorithm implemented the by TCPEP, which implements a slow start mechanism that imitates the TCP Reno congestion avoidance phase (thus resulting very slow, see Appendix C and [13] for further reading);
in all of the longer transfers (Figures 4.26, 4.27, 4.28, 4.29), the algorithm seems to perform very well: for each of the performed tests, the packet loss probability doesn’t affect at all the transfer time. By the way, if we give a look to the curves obtained for different error probability values, those ones look very different among them. A possible explanation to this phenomena may be given observing that for each throughput drop, we immediately have a spike of the same magnitude. This behaviour seems to be the demostration of the existence of some kind of loss-compensation algorithm;

• as a consequence of the first point, the system lasts a lot of time to achieve the maximum bitrate, in fact in the Downlink case, as it is set to a quite high value (i.e. 18Mbps), it doesn’t succeed to reach it at all.
Figure 4.26: Throughput curves obtained by performing the medium transfer in the Uplink direction with a packet error probability amount of respectively $10^{-2}$, $10^{-3}$ and $10^{-4}$.
Figure 4.27: Throughput curves obtained by performing the long transfer in the Uplink direction with a packet error probability amount of respectively $10^{-2}$, $10^{-3}$ and $10^{-4}$. 
Figure 4.28: Throughput curves obtained by performing the medium transfer in the Downlink direction with a packet error probability amount of respectively $10^{-2}$, $10^{-3}$ and $10^{-4}$
Figure 4.29: Throughput curves obtained by performing the long transfer in the Downlink direction with a packet error probability amount of respectively $10^{-2}$, $10^{-3}$ and $10^{-4}$
In order to make this algorithm the one with the best performance at all it is crucial to find a way to make the initial phase faster. A solution to this issue may be to split the transmission of the same content, thus setting up more flows in parallel. In our case, this solution has been tested by simply modifying the iperf command at the client side, specifying the number of parallel flows to be established using the \(-P\) option as follows (in our the simulations have been conducted establishing 8 parallel flows):

```
root@server:~# iperf -c CLIENT_IP -n 11M -i 5 -Z reno -P 8
root@client:~# iperf -s -P 8
```

This solution has been tested for the long and the medium transfers in both the Uplink and Downlink directions. The obtained results are shown in Figure 4.30 (for the Uplink) and Figure 4.31 (for the Downlink). In both the cases it is possible to observe that the throughput tends to the maximum obtainable values really faster than in the single flow cases.
This improvement is easily notable in the Uplink graphs, but the most satisfying result is the one obtained for the long transfer in the Downlink direction, in which the throughput finally succeeds in achieving its maximum value.

On the other hand this approach can be adopted only in those cases in which the servers use the HTTP/1.1 release, according to which a connection may be used for one or more request/response exchanges. The last release, HTTP/2, has removed this feature, thus making this kind of approach suitable for a quite restricted range of cases.

So, in the end we can conclude this section saying that the TCPeP algorithm demonstrates that the use of FEC techniques can be very helpful in making a system robust to losses. The only aspect that needs to be improved is the one related to the first part of the connection, that might be improved deploying High Speed TCP (HSTCP) congestion control approaches.

Figure 4.31: Throughput curves obtained by performing the medium and the long transfers in the Downlink using 8 parallel flows
4.3 Results Comparison

4.3.1 Numerical results

In this section, the results that have been shown so far are presented from three different points of view.

1) *Average throughput experienced*

The tables reported in Figure 4.32 are coherent with respect to what we stated so far:

- the throughput obtained in the all TCP Reno scenario comply quite good with what envisaged by the Mathis equation, according to which the maximum achieveable throughput is divided by $\sqrt{10}$ as well as the packet loss rate is increased of one order of magnitude. For this reason, this system results to perform well only in case of poorly lossy environments;

- the PEPsal algorithm results to be the best one in all of the cases when packet loss probability is not too large. In those cases, instead, PEPsal performances dramatically drop down.

- the TCPeP results show off the robustness of this algorithm with respect to losses, even though it results to be better performant in environments with limited channel capacity as a consequence of the slowness of the mechanism that it deploys at the beginning of the connection.

2) *Transfer times*

The time results in Figure 4.33 basically show the same aspects that have been highlighted by the previously commented ones, but also allow having a better focus of what the consequences of using one mechanism rather than the other one might be. More expesically, these tables allows us keeping in mind that using a pure TCP Reno approach in environments in which the probability of having losses is consistent is a way too high risk, as it might also decuplicate the time needed for doing the same transfer using other algorithms.

Moreover, also the PEPsal algorithm doesn’t seem to be in such a convenient situation, as it does have good performances also over quite lossy links, but the drop suffered by these performances represents a huge risk for the whole system safety.
Figure 4.32: Data collection of the average throughputs experienced during all of the executed tests
Figure 4.33: Data collection of the transfer times experienced during all of the executed tests
3) *Improvements obtained with respect to the context with no proxies deployed* (Figure 4.34).

The results exposed in Figure 4.34 show that in any case the implementation of one of the two tested PEPs succeeds in improving the overall performances, with some exceptions in cases of poorly lossy scenarios. Moreover, we can see that these improvements dramatically increase with the error probability in both cases in the Uplink direction. As regards the Downlink, instead, we can see that while the TCPeP behaves as good as in the Uplink, the PEPsal algorithm suffers more when losses become bigger. This aspect may suggest us that the PEPsal struggles more than the TCPeP in recovering the data rate in case of losses when the channel capacity is bigger.
4.3.2 Video streaming comparison

As a final test, the three configurations (i.e. all TCP Reno, PEPsal and TCPeP) analyzed in the previous paragraph have been used to perform the download of a video stored in the remote virtual machine.

First of all, we need to change the TCP congestion control algorithm used by our operating system. We do this by editing the file `/proc/sys/net/ipv4/tcp_congestion_control` so that to set it to the Reno algorithm:

```bash
echo "reno" > /proc/sys/net/ipv4/tcp_congestion_control
```

Then, we need to start the PEPs, if necessary. In the end, according to the procedure explained in chapter 3, once the nginx application has been started in the server, all we need is launching the browser from the local host terminal, typing the command:

```bash
local@localhost:~$ firefox http://192.168.240.2/klynt.mp4
```

We have tested the three cases with packet error probability values of $10^{-2}$, $10^{-3}$ and $10^{-4}$. As regards the order in which these tests have been conducted, the same order of the previous section has been followed.

So, we have started with the end-to-end TCP Reno scenario: in this case it has been noticed that the streaming of the video content is unacceptably slow with both $10^{-2}$ and $10^{-3}$ of packet error rate. In fact, as shown in the screenshots reported in Figure 4.35 and 4.36, the browser resulted to be unable to reproduce the video without being interrupted for more than one second. Only in case of very low error probability ($10^{-4}$), the algorithm results to be able to let the browser accumulate enough data to have a continuous reproduction (Figure 4.37).

Things behave in a slightly different way when the PEPsal scenario is mounted: the algorithm has the same performances of the previous case when the error probability is high ($10^{-2}$), as reported by the screenshot in Figure 4.38. On the other hand, this algorithm already shows its enhancing capability with the medium error probability value. As shown in Figure 4.39, in fact, the algorithm allows the client to download the whole video in a very few time.

Finally, it would have been very interesting to test the behaviour of the TCPeP algorithm, as, according to the results commented in the previous section, it should have been the one showing off the best performances in the highly lossy case. By the way, this test has not been performed as, despite the correct configuration of the whole scenario, when we have tried to test it, the TCP server proxy seemed to be unable to intercept packets and redirect them to the port the TCPeP application was listening to. As a result, the traffic captured by the satellite bridge was encapsulated over TCP, instead of UDP as it should result after the TCPeP codification process.
CHAPTER 4. EXPERIMENTAL SETUP

Figure 4.35: Screenshot done in the end-to-end Reno scenario with $10^{-2}$ packet error probability

Figure 4.36: Screenshot done in the end-to-end Reno scenario with $10^{-3}$ packet error probability

Figure 4.37: Screenshot done in the end-to-end Reno scenario with $10^{-4}$ packet error probability
CHAPTER 4. EXPERIMENTAL SETUP

Figure 4.38: Screenshot done in the PEPsal scenario with $10^{-2}$ packet error probability

Figure 4.39: Screenshot done in the PEPsal scenario with $10^{-3}$ packet error probability
Chapter 5

Conclusions and future developments

In this chapter a series of conclusions on the developed work (Section 2.1), with the suggestion for some future developments (Section 2.2) are presented.

5.1 Conclusions

In this work it has been investigated about problems related to the deployment of streaming services that require very high real time infrastructures performances. As said in the introduction chapter, this is an issue that has increased in importance in the very last years. The most important challenge has been launched to us by those providers that were based on a well consolidated platform and that now need a complete change of paradigm in order to be able to provide real time streaming services. It’s the case for example of YouTube, which had always its platform on the usage of the TCP protocol, and that only since a little time ago has also started provide live streaming services.

Even if it had a very specific goal, that is testing some of the solutions that have been proposed so far to this kind of problem, the approach that we have given to this work can be subdivided in two phases. The first phase has mainly been focused on studying and getting consciousness about the VNX tool has been designed for Linux system that allow creating and simulating virtual network over a single physical machine running a Linux operating system. Then, a series of example tests have been performed in order to understand the way the tools to be used to evaluate the final scenario work. After this part, the methodologies used in this field and the algorithms that were intended to be evaluated have been studied. Once finished this theoretical study all the tests needed for the performance evaluation have been done.
So it is possible to conclude saying that the analysis and the evaluations that have been done in this work have given a flavor of how various can be the approaches that can be adopted to address the described issues. As a matter of fact, the solutions that have been studied are just some particular cases of the split-connection methodology, which also is one of the possible approaches that can be adopted in this situation.

5.2 Future developments

Even PEP algorithms that have better performances in satellite networks already exist, according to the studies conducted and the results obtained with the algorithms that have been present in this work may allow us to do some considerations about approaches that can be adopted to obtain improvements on these algorithms. In particular, it is possible to notice that the observations that have been done in the previous chapter basically lead to the conclusion according to which combining the techniques used for the PEPsal and the TCPeP algorithms may be the way to obtain an optimally performing algorithm.

As a matter of fact, it has been reported that thanks to its aggressive congestion control mechanism, the PEPsal algorithm allows obtaining very good throughput performances, but, on the other hand, it has also been shown how it is affected by scarce robustness in presence of highly lossy links. On the contrary, the TCPeP performs very well in terms of robustness against losses thanks (as a consequence of implementing selective acknowledgment, instead of forcing the sender to retransmit all the data from the loss event like the baseline TCP does), but has very strong limits in the slow start phase, in which it basically implements the TCP Reno congestion avoidance mechanism, thus resulting very slow. As a result, the TCPeP penalizes a lot those transfers that last less than thirty seconds, especially in situations in which losses don’t represent at all a big problem.

An objective for eventual future research work may be trying to combine these two approaches, in a way to obtain the best performance possible.
Appendices
Appendix A

xml codes used to generate the VNX scenarios

A.1 Basic two-hosts scenario

<?xml version="1.0" encoding="UTF-8"?>

<!-- Description: As simple tutorial scenario made of 1 LXC Ubuntu virtual machine connected to

<vnx xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="/usr/share/xml/vnx/vnx-2.00.xsd">
    <global>
        <version>2.0</version>
        <scenario_name>lorenzo1</scenario_name>
        <ssh_version>2</ssh_version>
        <ssh_key>~/.ssh/id_rsa.pub</ssh_key>
        <automac/>
        <vm_mgmt type="net" network="172.16.31.0" mask="24" offset="10">
            <mgmt_net net="lxcbr0" managed="no" config="manual"/>
        </vm_mgmt>
        <vm_defaults>
            <console id="0" display="no"/>
            <console id="1" display="yes"/>
        </vm_defaults>
    </global>
</vnx>
<net name="vbridge" mode="virtual_bridge" />
<vm name="remote" type="lxc" arch="x86_64">
    <filesystem type="cow">/usr/share/vnx/filesystems/rootfs_lxcUbuntu64</filesystem>
    <if id="1" net="vbridge">
        <ipv4>192.168.1.2/24</ipv4>
    </if>
    <route type="ipv4" gw="172.16.31.1">default</route>
</vm>
<host>
    <hostif net="vbridge">
        <ipv4>192.168.1.1/24</ipv4>
    </hostif>
</host>
</vnx>

A.2 TCPeP trial scenario

<?xml version="1.0" encoding="UTF-8"?>
<!--Description: A simple tutorial scenario made of 2 LXC Ubuntu virtual machines (a remote host and a Performance-enhancing Proxy) connected to the physical host through two virtual networks. -->
<vnx xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="/usr/share/xml/vnx/vnx-2.00.xsd">
    <global>
        <version>2.0</version>
        <scenario_name>lorenzo_2</scenario_name>
    </global>
    <ssh_version>2</ssh_version>
    <ssh_key>"/.ssh/id_rsa.pub"</ssh_key>
    <automac/>
    <!--vm_mgmt type="none" -->
    <!--vm_mgmt type="private" network="10.250.0.0" mask="24" offset="200">-->
    <host_mapping />
</vm_mgmt-->
    <vm_mgmt type="net" network="172.16.31.0" mask="24" offset="10">
        <mgmt_net net="lxcbr0" managed="no" config="manual"/>
    </vm_mgmt>
    <vm_defaults>
        <console id="0" display="no"/>
<console id="1" display="yes" />
</vm_defaults>
<!-- cmd-seq seq="ls12">ls1,ls2</cmd-seq -->
<cmd-seq seq="ls123">ls12,ls3</cmd-seq>
<cmd-seq seq="ls1234">ls123,ls4</cmd-seq -->
<!-- vnx_cfg> tutorial_lxc_ubuntu.cvnx</vnx_cfg -->
</global>
<net name="feeder" mode="virtual_bridge" />
<net name="sinker" mode="virtual_bridge" />
<vm name="pep2" type="lxc" arch="x86_64">
<filesystem type="cow">/usr/share/vnx/filesystems/rootfs_lxc_ubuntu64</filesystem>
<if id="1" net="feeder">
<ipv4>192.168.1.2/24</ipv4>
</if>
<if id="2" net="sinker">
<ipv4>192.168.2.1/24</ipv4>
</if>
<route type="ipv4" gw="172.16.31.1">default</route>
<forwarding type="ip" />
</vm>
<vm name="sinksrv2" type="lxc" arch="x86_64">
<filesystem type="cow">/usr/share/vnx/filesystems/rootfs_lxc_ubuntu64</filesystem>
<if id="1" net="sinker">
<ipv4>192.168.2.2/24</ipv4>
</if>
<route type="ipv4" gw="172.16.31.1">default</route>
<route type="ipv4" gw="192.168.2.1">192.168.1.0/24</route>
</vm>
<host>
<hostif net="feeder">
<ipv4>192.168.1.1/24</ipv4>
</hostif>
<route type="ipv4" gw="192.168.1.2">192.168.2.0/24</route>
</host>
APPENDIX A. XML CODES USED TO GENERATE THE VNX SCENARIOS

A.3 Complete scenario

<vnx xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="/usr/share/xml/vnx/vnx-2.00.xsd">
<global>
  <version>2.0</version>
  <scenario_name>simple_lorenzo</scenario_name>
  <automac_offset="4"/>
  <vm_mgmt type="none"/>
  <vm_defaults>
    <console id="0" display="no"/>
    <console id="1" display="yes"/>
  </vm_defaults>
</global>
<help>
  <seq_help seq='pep_start'>Starts pepsal with cubic in eth2 on pep vm</seq_help>
  <seq_help seq='pep_stop'>Stops pepsal on pep vm</seq_help>
  <seq_help seq='pep_status'>pepsal status on pep vm</seq_help>
    <seq_help seq='pep_cfg'>Loads pep.cfg file from HOST to pep 's</seq_help>
  <seq_help seq='check_eot'>Checks End of Transmission on pep 's</seq_help>
  <seq_help seq='queue_start'>Starts queue sampling on hub and saterm vm' s</seq_help>
  <seq_help seq='queue_stop'>Stops queue sampling on hub and saterm vm' s</seq_help>
  <seq_help seq='queue_status'>queue status on hub and saterm vm' s</seq_help>
  <seq_help seq='iperf_start'>Starts iperf server on sink</seq_help>
  <seq_help seq='iperf_stop'>Stops pepsal server on sink</seq_help>
    <seq_help seq='iperf_cfg'>Loads iperf config file on sink</seq_help>
  <seq_help seq='nginx_start'>Starts nginx server on sink</seq_help>
  <seq_help seq='nginx_stop'>Stops nginx server on sink</seq_help>
    <seq_help seq='on_boot'>Loading of filestrees and files on boot</seq_help>
</help>
<global>

<net name="feeder" mode="virtual_bridge" />
<net name="satbr" mode="virtual_bridge" />
<net name="satellite" mode="virtual_bridge" />
<net name="hubbr" mode="virtual_bridge" />
<net name="remotebr" mode="virtual_bridge" />
<net name="virbr0" mode="virtual_bridge" managed="no" />

<vm name="satpep" type="lxc" arch="x86_64">
  <filesystem type="cow">/usr/share/vnx/filesystems/rootfs_lxc_ubuntu64</filesystem>
  <if id="3" net="virbr0">
    <ipv4>dhcp</ipv4>
  </if>
  <if id="1" net="feeder">
    <ipv4>192.168.200.2/24</ipv4>
  </if>
  <if id="2" net="satbr">
    <ipv4>192.168.210.1/24</ipv4>
  </if>
  <route type="ipv4" gw="192.168.210.2">192.168.220.0/24</route>
  <route type="ipv4" gw="192.168.210.2">192.168.230.0/24</route>
  <route type="ipv4" gw="192.168.210.2">192.168.240.0/24</route>
  <forwarding type="ip" />
  <filetree seq="on_boot" root="/tmp/">files/satpep/progs</filetree>
  <filetree seq="on_boot" root="/sbin/">files/common/sbin</filetree>
  <filetree seq="on_boot" root="/etc/pepsal/">files/satpep/etc</filetree>
  <filetree seq="pep_cnf" root="/tmp/progs/">files/satpep/progs/pep.cnf</filetree>
  <exec seq="on_boot" type="file" ostype="system"> files/satpep/setup/fq.sh</exec>
  <exec seq="on_boot" type="file" ostype="system"> files/satpep/setup/ip.sh</exec>
  <exec seq="pep_start" type="verbatim" ostype="exec"> nohup /tmp/progs/
APPENDIX A. XML CODES USED TO GENERATE THE VNX SCENARIOS

pep.sh start</exec>
<exec seq="pep_stop" type="verbatim" ostype="exec">nohup /tmp/progs/pep.sh stop</exec>
<exec seq="pep_status" type="verbatim" ostype="exec">nohup /tmp/progs/pep.sh status</exec>
<exec seq="check_eot" type="verbatim" ostype="system">/tmp/progs/check_EOT.sh</exec>
</vm>

<vm name="satterm" type="lxc" arch="x86_64">
<filesystem type="cow">/usr/share/vnx/filesystems/rootfs_lxc_uubunta64</filesystem>
<if id="3" net="virbr0">
<ipv4>dhcp</ipv4>
</if>
<if id="1" net="satbr">
<ipv4>192.168.210.2/24</ipv4>
</if>
<if id="2" net="satellite">
<ipv4>192.168.220.1/24</ipv4>
</if>
<route type="ipv4" gw="192.168.210.1">192.168.200.0/24</route>
<route type="ipv4" gw="192.168.220.2">192.168.230.0/24</route>
<route type="ipv4" gw="192.168.220.2">192.168.240.0/24</route>
<forwarding type="ip" />
</vm>

<vm name="hub" type="lxc" arch="x86_64">
<filesystem type="cow">/usr/share/vnx/filesystems/rootfs_lxc_uubunta64</filesystem>
<if id="3" net="virbr0">
<ipv4>dhcp</ipv4>
</if>
<if id="1" net="satellite">
<ipv4>192.168.220.2/24</ipv4>
</if>
APPENDIX A. XML CODES USED TO GENERATE THE VNX SCENARIOS

```xml
<if id="2" net="hubbr">
   <ipv4>192.168.230.1/24</ipv4>
</if>

<route type="ipv4" gw="192.168.220.1">192.168.200.0/24</route>
<route type="ipv4" gw="192.168.220.1">192.168.210.0/24</route>
<route type="ipv4" gw="192.168.230.2">192.168.240.0/24</route>
<forwarding type="ip"/>

<vm name="pepm" type="lxc" arch="x86_64">
   <filesystem type="cow">/usr/share/vnx/filesystems/rootfs_lxc_ubuntu64</filesystem>
   <if id="3" net="virbr0">
      <ipv4>dhcp</ipv4>
   </if>
   <if id="1" net="hubbr">
      <ipv4>192.168.230.2/24</ipv4>
   </if>
   <if id="2" net="remotebr">
      <ipv4>192.168.240.1/24</ipv4>
   </if>
   <route type="ipv4" gw="192.168.230.1">192.168.200.0/24</route>
   <route type="ipv4" gw="192.168.230.1">192.168.210.0/24</route>
   <route type="ipv4" gw="192.168.230.1">192.168.220.0/24</route>
   <forwarding type="ip"/>
   <filetree seq="on_boot" root="/tmp/">files/pep/progs</filetree>
   <filetree seq="on_boot" root="/sbin/">files/common/sbin</filetree>
   <filetree seq="on_boot" root="/etc/pepsal/">files/pep/etc</filetree>
   <filetree seq="pep_cfg" root="/tmp/progs/">files/pep/progs/pep.cfg</filetree>
   <exec seq="on_boot" type="file" ostype="system">files/pep/setup/fq.sh</exec>
   <exec seq="on_boot" type="file" ostype="system">files/pep/setup/ip.sh</exec>
   <exec seq="pep_start" type="verbatim" ostype="exec">nohup /tmp/progs/
```
APPENDIX A. XML CODES USED TO GENERATE THE VNX SCENARIOS

pep.sh start</exec>
<exec seq="pep_stop" type="verbatim" ostype="exec">nohup /tmp/progs/pep.sh stop</exec>
<exec seq="pep_status" type="verbatim" ostype="exec">nohup /tmp/progs/pep.sh status</exec>
<exec seq="check_eot" type="verbatim" ostype="system">/tmp/progs/check_EOT.sh</exec>

</vm>

<vm name="remote" type="lxc" arch="x86_64">
  <filesystem type="cow">/usr/share/vnx/filesystems/rootfs_lxcUbuntu64</filesystem>
  <if id="2" net="virbr0">
    <ipv4>dhcp</ipv4>
  </if>
  <if id="1" net="remotebr">
    <ipv4>192.168.240.2/24</ipv4>
  </if>
  <route type="ipv4" gw="192.168.240.1">192.168.200.0/24</route>
  <route type="ipv4" gw="192.168.240.1">192.168.210.0/24</route>
  <route type="ipv4" gw="192.168.240.1">192.168.220.0/24</route>
  <route type="ipv4" gw="192.168.240.1">192.168.230.0/24</route>
  <filetree seq="on_boot" root="/usr/local/">files/snksrv/nginx</filetree>
    <exec seq="on_boot" type="file" ostype="system">files/snksrv/setup/sink_rcv_setup.sh</exec>
    <!--exec seq="iperf_start" type="verbatim" ostype="system">/tmp/iperf/iperf.sh</exec>
    <exec seq="iperf_stop" type="verbatim" ostype="system">/tmp/iperf/iperf.sh stop</exec>
    <exec seq="nginx_start" type="verbatim" ostype="system">/usr/local/nginx/sbin/nginx</exec>
    <exec seq="nginx_stop" type="verbatim" ostype="system">/usr/local/nginx/sbin/nginx -s stop</exec>
  </if>
</vm>
<host>
  <hostif net="feeder">
    <ipv4>192.168.200.1/24</ipv4>
  </hostif>
  <route type="ipv4" gw="192.168.200.2">192.168.210.0/24</route>
  <route type="ipv4" gw="192.168.200.2">192.168.220.0/24</route>
  <route type="ipv4" gw="192.168.200.2">192.168.230.0/24</route>
  <route type="ipv4" gw="192.168.200.2">192.168.240.0/24</route>
</host>
</vnx>
Appendix B

List of commands used to realize the performed test

B.1 Setting the lossy link and the channel capacity

root@saterm:~# tc qdisc replace dev eth1 root handle 1:0 estimator 250ms 361ms netem delay 285ms limit 3000 loss random <x>%;
root@saterm:~# tc qdisc replace dev eth2 root handle 1:0 estimator 250ms 361ms tbf rate 2000kbit burst 6120 lat 570ms mtu 1540;
root@hub:~# tc qdisc replace dev eth1 root handle 1:0 estimator 250ms 361ms tbf rate 18000kbit burst 6120 lat 570ms mtu 1540;
root@hub:~# tc qdisc replace dev eth2 root handle 1:0 estimator 250ms 361ms netem delay 285ms limit 3000 loss random <x>%;

B.2 Iperf file transfers

Uplink direction

root@remote:~# iperf -s
root@phyhost:~# iperf -c 192.168.240.2 -n 200000000 -i 5 -Z reno -F <filename>

Downlink direction

root@phyhost:~# iperf -s
root@remote:~# iperf -c 192.168.200.1 -n 200000000 -i 5 -Z reno -F <filename>
Appendix C

The TCPeP congestion control algorithm

/* The function is called upon the reception of an ACK */
if (state->slowStartMode) {
    // While we are in slow-start mode, increment at each event
    state->congestionWindow += 1;
    if (state->congestionWindow > SS_THRESHOLD) {
        state->slowStartMode = false;
    }
} else {
    // Congestion avoidance mode
    delta = 1 - (state->longTermRttAverage / state->shortTermRttAverage);
    if (delta < ALPHA) {
        // Increase the window
        state->congestionWindow += (INCREMENT / state->congestionWindow);
    } else if (delta > BETA) {
        // Decrease the window
        state->congestionWindow -= (INCREMENT / state->congestionWindow);
    }
    // If delta is in between, do not change the window
}
Bibliography


[5] A cross-layer architecture for TCP splitting in the forward channel of satellite networks, Juanjo Alins, Jorge Mata-Diaz, Jose L. Muoz, Elisabeth Rendn-Morales, Oscar Esparza, Universitat Politcnica de Catalunya (UPC), Spain, Department of Telematics Engineering;

[6] Applying Network Coding to TCP, Leonardo Andres Urbina Tovar, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 2012;

[7] Notes on Reed-Solomon codes, Mark Haiman;

[8] Evolucion de VNUML a la herramienta de gestion de escenarios de red virtuales multiplataforma VNX, Jorge Eusebio Somavilla Franco, David Fernandez Cambronero, Madrid 2013;


[10] TCP Hybla: a TCP enhancement for heterogeneous networks, Carlo Caini, Rosario Fimrincieli, Bologna University


[12] Binary increase congestion control (BIC) for fast long-distance networks, Lisong Xu, K. Harfoush, Injong Rhee, Department of Computer Science, North Carolina State University, Raleigh, NC, USA
Glossary of Acronyms

AIMD Additive Increase Multiplicative Decrease
awnd Advertised Window
BDP Bandwidth-Delay Product
CTCP Coded TCP
cwnd Congestion Window
DUPACK Duplicate Acknowledgment
DVB-S2 Digital Video Broadcasting - Satellite 2nd generation
DVB-RCS Digital Video Broadcasting - Return Channel Satellite
EWMA Exponentially Weighted Moving Average
FEC Forward Error Correction
HSPA High-Speed Packet Access
HSTCP High-Speed TCP
HTTP Hypertext Transfer Protocol
IP Internet Protocol
ISO International Organization for Standardization
LAN Local Area Network
LTE Long Term Evolution
LTE-A Long Term Evolution Advanced
LXC LinuX Container
MAC Medium Access Control
MSS Maximum Segment Size
MTU Maximum Transmission Unit
NAT Network Address Translation
OSI Open Systems Interconnection
PEP Performance-Enhancing Proxy
RFC Request For Comment
RTP Real-time Transport Protocol
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTCP</td>
<td>Real-time Transport Control Protocol</td>
</tr>
<tr>
<td>RTSP</td>
<td>Real-time Transport Streaming Protocol</td>
</tr>
<tr>
<td>RTT</td>
<td>Round Trip Time</td>
</tr>
<tr>
<td>SACK</td>
<td>Selective Acknowledgment</td>
</tr>
<tr>
<td>SMS</td>
<td>Short Message Service</td>
</tr>
<tr>
<td>SNC</td>
<td>Sparse Network Coding</td>
</tr>
<tr>
<td>SSH</td>
<td>Secure Shell</td>
</tr>
<tr>
<td>TBF</td>
<td>Token Bucket Filter</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>UML</td>
<td>User Mode Linux</td>
</tr>
<tr>
<td>UMTS</td>
<td>Universal Mobile Telecommunication System</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>VNUML</td>
<td>Virtual Network User Mode Linux</td>
</tr>
<tr>
<td>VNX</td>
<td>Virtual Networks over linuX</td>
</tr>
<tr>
<td>VPN</td>
<td>Virtual Private Network</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>
Acknowledgements

At the end of this work, I would express my gratefulness to this project advisor, professor Jorge Mata Diaz, which has always been helpful and available, and has always encouraged me to work hard without being concerned about my knowledge gaps regarding the tools I was supposed to work with. I’ve really appreciated it, especially because it has allowed me to always stay focused on the work I had to do, without feeling the anxiety of being too slow or poorly experienced. Then I would also express my gratitude to my girlfriend, Alessia, my mother and my father, Nunzia and Antonio, my brothers, Gabriele and Giulio, the rest of my family and all of my friends who have supported me during these amazing years.