UNIVERSITÀ DI PISA
Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Estimating the Capacity of the Links along an Internet Path via an Android Smartphone

Relatori:
Prof. Luciano Lenzini
Ing. Alessio Vecchio
Prof. Enrico Gregori

Candidato:
Angelo Favale

Anno accademico 2013-2014

“While knowledge can create problems,

it is not through ignorance that we can solve them.”

Isaac Asimov

Abstract

A lot of capacity estimation tools have been designed and developed during the last years. This measurement is important in many Internet applications and protocols. In particular, it is an appealing information for end system multicast and overlay network configuration protocols ([12, 13, 14]), content location and delivery in peer-to-peer (P2P) networks ([15, 16]), network aware cache and replica placement policies ([17, 18]). Eventually, the estimation of this parameter can also be useful for network operators. There are a lot of tools that estimate the capacity of the bottleneck, but there are few tools that estimate the capacity of each link along a path. Furthermore, most of them are designed to work in a wired network. In this thesis we focus on the development of a tool able to discover the capacity of each link in an Internet path via an Android smartphone. In particular, we use an approach similar to that described in [1], indeed we use an active technique based on an end-to-end measurement. The difference is that our tool is an application context-aware, because it takes into account the connection to Internet of the smartphone. The Android smartphone can connect to the Internet through different network technologies (i.e. Wifi, GSM, UMTS, LTE, ...). This network technologies affect the parameters of estimation of our tool. The use of smartphones to take measurement of the network is an approach also used by PORTOLAN. The PORTOLAN PROJECT ([10, 11]) is an Internet measurement system that aims at both discovering the Internet graph, using traceroute, and building maps of signal coverage through smartphone-based crowdsourcing. Portolan is developed by the department of Computer Engineering of the University of Pisa and the Institute of Information of Technology of the Italian National Research Council (IIT/CNR).

Contents

List of Figures

Chapter 1

Introduction

In this introduction chapter we give a description of the objective of this thesis showing the environment where it works and the state of the art.

The chapter is structured in three parts. Firstly, we present a brief description of Internet, its history and topology. Secondly, we describe the objective of this thesis. Finally, we show the state of the art of the capacity estimation tools.

1.1 A brief history of Internet

The nowadays Internet took a lot of time to born. It all started with some researches of the 1960s at DARPA (Defense Advanced Research Projects Agency), firstly known simply as ARPA [8]. The U.S. government in collaboration with private commercial interests commissioned these researches in order to build robust, fault-tolerant and distributed networks between computers.

In 1965 the TX-2 computer in Massachusetts was connected to the Q32 in California with a low speed dial-up telephone line creating the first (however small) wide-area computer network ever build. This experiment showed the feasibility of the time-shared computers to work together, and that was necessary a packet switched network rather than the circuit switched telephone system.

The first packet switched network was developed in 1969, it was called ARPANET (Advanced Research Projects Agency NETwork). This network connected four major computers at UCLA (University of California, Los Angeles), Standford Research Institute, UCSB (University of California, Santa Barbara) and the University of Utah. Figure 1.1 shows a 4-node ARPANET diagram.

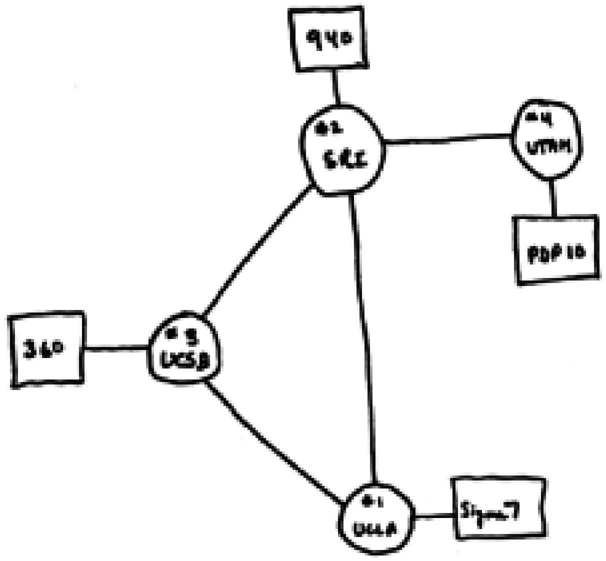


Figure 1.1: 4-node ARPANET diagram

ARPANET was funded by the U.S. Department of Defense and universities. It was designed to provide a communications network that would work even if some of the major sites were down. If the most direct route was not available, routers would direct traffic around the network via alternate routes.

The early Internet was used by computer experts, engineers, scientists, and librarians. There was nothing friendly about it. There were no home or office personal computers in those days, and anyone who used it, whether a computer professional or an engineer or scientist or librarian, had to learn to use a very complex system.

The funding of a new U.S. backbone by the National Science Foundation in the 1980s, as well as private funding for other commercial backbones, led to worldwide participation in the development of new networking technologies, and the merger of many networks. Commercial Internet Service Providers (ISPs) began to emerge in the late 1980s and early 1990s, and in 1995 the removing of the last restrictions to carry commercial traffic led to the Internet commercialization.

The availability of pervasive networking (i.e., the Internet) along with powerful affordable computing and communications in portable form (i.e., laptop computers, PDAs, cellular phones), is the cause of the impressive exponential growth of both the links added everyday and the number of users in the world.

1.2 The current Internet

The current Internet is a *network of networks* that consists of millions of private, public, academic, business, and government networks, of local to global scope, that are linked by a broad array of electronic, wireless and optical networking technologies.

In particular it is a global system of interconnected computer networks based on the TCP/IP (Transport Control Protocol/Internet Protocol) standard Internet protocol suite. It serves several billion users worldwide. More than 100 countries are linked into exchanges of data, news and opinions. According to Internet World Stats, as of June 30, 2012 there was an estimated 2,405,518,376 Internet users worldwide. This represents 34.3% of the world's population.

A this moment we know that the core of Internet is structured in a multi-tier hierarchy of IP [9] transit providers that are connected because of commercial and business agreements. Figure 1.2 shows the actual Internet structure.

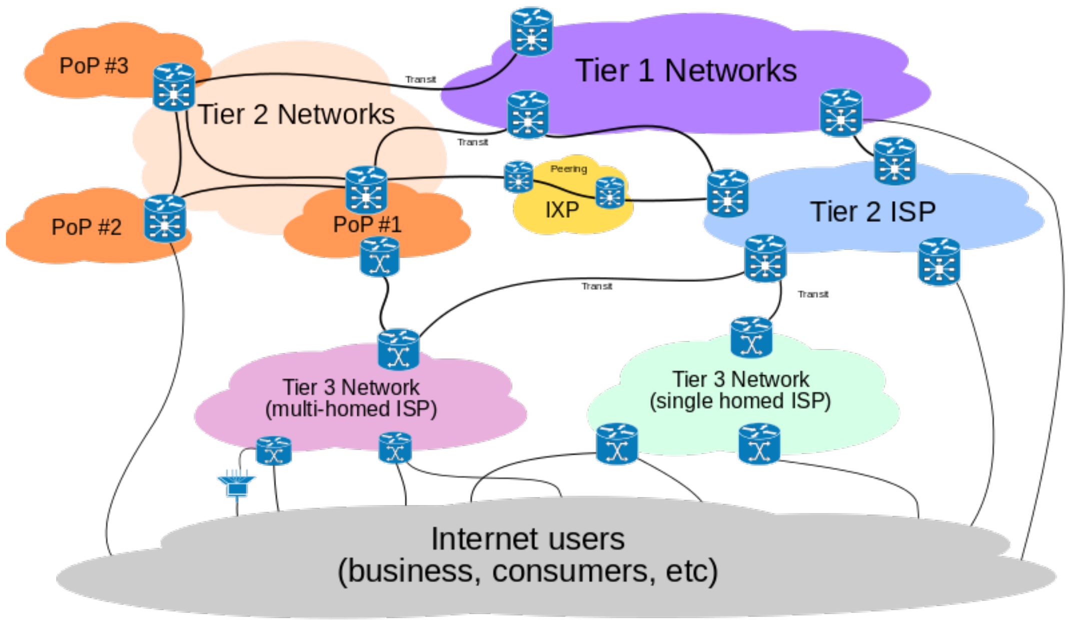


Figure 1.2: The current Internet core structure

The previous figure only shows the structure of the core of the network, but the entire Internet graph remain undiscovered.

Knowing the entire Internet graph would be very important in many fields like:

- development and improvement of the network protocols
- business relationship between ISPs
- financial investments
- containment strategy for virus
- distribution of the digital content service provider
- analysis of the Internet fault-tolerance
- socio-economic studies on the development and coverage of the Internet

The problem is that discovering the entire Internet topology is a very difficult task due to several key features of network development. these characteristics are:

- size and growth of Internet
- heterogeneity of how to administer the networks within the Internet
- evolutionary principles dictated by the tradeoff between cooperation and competition among ASs (Autonomous System)

Many measurement systems use a top-down approach to discover the Internet topology, that is, from the Internet core to its edges.

The PORTOLAN INTERNET TOPOLOGY MEASUREMENT SYSTEM instead uses an orthogonal approach. It is based on a bottom-up and bottom-to-bottom approach, that is, from the edge to the Internet core and from edge to edge (between end users) respectively.

1.3 Portolan Internet Topology Measurement System

The Portolan system is a Internet measurement system designed to discover the Internet topology at the autonomous system level and to build the map of the signal coverage. Its most important characteristic is that it is based on a smartphone-based crowdsourcing.

Indeed, in this system we have a client side, an Android application installed on several smartphones, and a server side. When these smartphones are connected to the Internet via any access network (i.e. Wi-fi, GSM, UMTS, LTE, ...) the tool can perform its measurements. It is important to highlight the fact that the data collected come from the smartphone's volunteers. If there are many volunteers running this application this tool can collect a lot of useful data.

This kind of approach allows Portolan to be focused on the analysis of the Internet edge. At this moment the existing measurement tools are not focused on this portion of the Internet. This make the Portolan system an appealing measurement system to discover new features of the current Internet, as described in [10, 11].

The Portolan system is developed by the University of Pisa and IIT/CNR.

1.4 Objective of this thesis

In this thesis we are not focused in discovering the Internet topology but we are interested in analyzing the feasibility to measure the capacity of the links in Internet.

In particular we develop a tool for estimating the capacity of the links along an Internet path via an Android smartphone.

To retrieve these informations, we use a very similar approach to that adopted in the Portolan system. Indeed, the estimation is performed using, on client side, an Android smartphone.

This measurement is important in many Internet applications and protocols.

End system multicast and overlay network configuration protocols ([12, 13, 14]) knowing the capacity of the links can use a capacity-aware path to deliver the packets.

Content location and delivery in peer-to-peer (P2P) networks ([15, 16]) strongly depends on the characteristics of this scenario. An important parameter that can be useful for these protocols is the capacity of the links in this network.

Another related application that can take advantage on the knowledge of this parameter is the network aware cache and replica placement policies ([17, 18]).

Eventually, an accurate measurement of the capacities within a network can be exploited by network operators concerned with problems such as capacity provisioning and traffic engineering.

1.5 State of the art

In these years have been developed a lot of capacity estimation tool. These tools can be classified in different ways depending on the property that you want to highlight.

The main differences between these tools are:

- active VS passive technique

  Most of the tools use an active technique. These tools send the probe packets for the sole purpose of making an estimation. Instead, the tools that use a passive technique take advantage of the packets sent by other applications to perform their estimation. In this way it is less intrusive rather than the tools that use active techniques. The disadvantage of the passive techniques is that in this case the tool depends on another application that must exchange packets.

- End-to-end VS hop-by-hop measurements

  End-to-end techniques based their estimation only on the reply of the end-host. Generally, these technique are only able to estimate the bottleneck of a path. Instead, hop-by-hop techniques take advantage of the ICMP (Internet Control Message Protocol) protocol. ICMP is one of the core protocols of the Internet Protocol Suite. It is used by the routers in the network to send error messages indicating, for example, that a service is not available or that a host or a router is unreachable. Hence, these tools use these messages to perform the estimations on every link in a path

These two classifications are orthogonal to each other, as one does not preclude the other. Indeed, for example, there are tools that are both active and end-to-end, and others tools that are both active and hop-by-hop.

It is important to highlight the fact that most of these tools are designed to work in wired networks.

When in the network there are wireless links, the estimation is more complicated. In this case wireless capacity estimation depends also on the topology, path layout, interference between nodes along the path and several other environmental parameters.

Among the most well-known tool for the capacity estimation we can remember Nettimer and Pathchar.

1.5.1 Nettimer

Nettimer is a tool to infer the capacity of the bottleneck in a path, as described in [6]. This tool can use both passive and active techniques, but its estimations are always performed in an end-to-end fashion.

It can simulate or passively collect network traffic, and can also actively probe the network using a packet-pair ’tailgating’ technique. There is no requirement for any special information from the network and no limitation to a particular transport protocol.

1.5.2 Pathchar

Pathchar is a tool that infers the characteristics of links along an Internet path, as described in [19, 20]. In particular it is able to estimate the latency, bandwidth, loss rate and queue delays experienced at each link.

This tool uses an active technique with hop-by-hop measurement. Indeed, this tool uses a similar approach to that used by traceroute ([21]).

Pathchar works by sending out a series of probes with increasing values of TTL and varying packet sizes. For each probe it measures the time until the error packet ICMP is received. By performing statistical analysis of these measurements, pathchar infers the latency and bandwidth of each link in the path, the distribution of queue times, and the probability that a packet is dropped.

Figure 1.3 shows the operation performed by pathchar when it uses a fixed packet size.

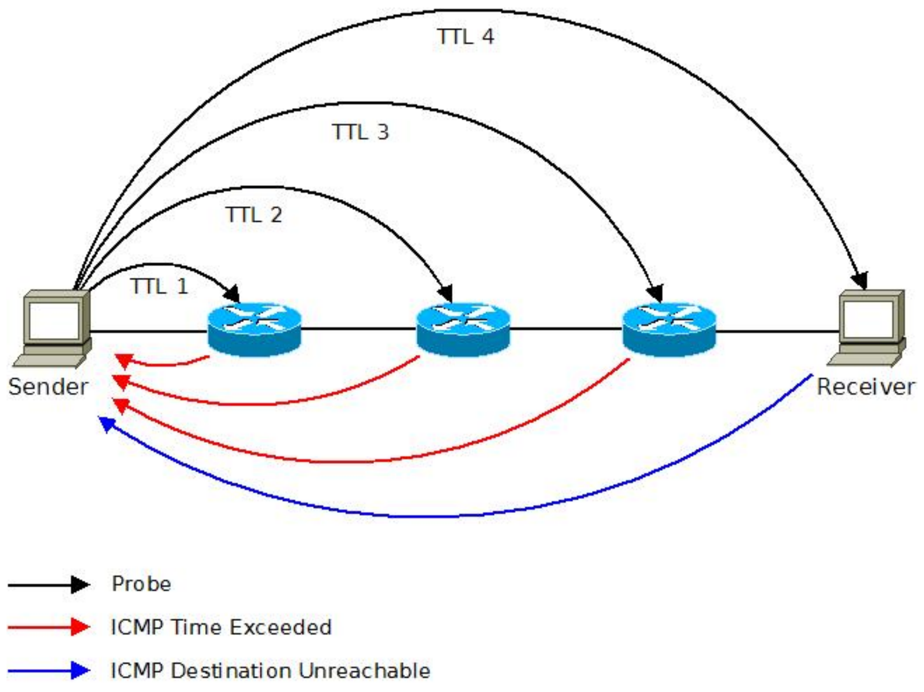


Figure 1.3: Pathchar example

Chapter 2

Algorithm

In this chapter we analyze how we can estimate the throughput of each link in a certain path.

The chapter is structured into three parts. Firstly, we describe the basic definitions and the assumptions used. Secondly, we explain the differences between the estimation of the first link in a path and the other links. Finally, we show the algorithm used, highlighting strengths and weaknesses.

2.1 Basic Definitions

In this section we describe basic constructs of our probing sequences and the corresponding terminology.

With the term *probe* we mean a sequence of one or more packets sent from a common origin.

The sequence of links crossed by the probes form a *path* and a subset of links like $L_1, \cdots, L_j$ with $j < n$ is called *prefix path*.

When a contiguous sequence of packets are transmitted in the same probe without time separation between each packet than we say that the probe is transmitted *back-to-back*.

In particular when the probe is formed by two packets sent back-to-back we call it a *packet-pair probe*, instead when the probe is constituted by three or more packets sent back-to-back then we call it a *packet-train probe*.

A *uniform* probe is one in which every packet has the same size; likewise, a *non-uniform* probe is made of packets with different sizes.

A packet is *hop limited* if its Time To Live (TTL) is set to a value such that this packet do not reach the destination.

The packets inside a probe are parametrized by:

s(p) its size in bits

D(p) its final destination

h(p) its maximum hop count

Every probing technique exploits these parameters to build different packets in a probe.

These probes are then used to infer some information about the path.

To denote a probe, we refer to each probe packet with a different lowercase letter, and represent the sequential order in which they are transmitted from the probing host by writing them from left to right.

We denote interpacket spacing with square brackets, while a sequence of identical probe packets sent back-to-back is represented by curly brackets with the number of packets as superscript. As an example, $[\,pm\,]\,[\,pm\,]\,[\,\{q\}^r\,]$ denotes transmission of identical two-packet probes followed by a sequence of r packets 'q' sent back-to-back.

The throughput of the bottleneck's path segment between the links i and j is denoted by $b_{i,j}$, while b_i stands for the throughput of the i-th link. In particular $b_{i,j} = \min\limits_{i \le k \le j} b_k$.

We use the term *interarrival time of packets p and m at link i* to indicate the time elapsed between the arrival of the last byte of p and the last byte of m at the link i-th. This is represented with Δ_i.

Similarly, we use the term *interdeparture time of packets p and m at link i* to denote the time elapsed between the transmission of the last byte of p and the transmission of the last byte of m at link i-th.

By these definitions we can say that the interarrival time of packets p and m at link i is equal to the interdeparture time of packets p and m at link i-1.

In a path L, composed by links $L_1, L_2, \cdots, L_n$, where the bottleneck is located at link i, the bottleneck's capacity is denoted by the following notation

$$b_i = \min_{1 \le k \le n} b_k$$

When we say that an estimation is reliable we mean that this estimation reproduces accurately the real value of the parameter we want to estimate.

2.2 Assumptions

Initially these assumptions are used in order to simplify the analysis of the environment where we work but in the end we remove them and check the results we obtain.

These assumptions are common to many probing studies (e.g., [3, 4, 5, 6]) and are enumerated below:

1. Routers are store-and-forward and use FIFO queueing.
2. Analytic derivations assume an environment free from cross-traffic.

The first assumption is needed to ensure that the probe packet orderings are preserved. Most of the routers in Internet can be represented by making this assumption.

The second assumption is used to make a simple analysis of the algorithm. In Internet this assumption is absolutely incorrect.

2.3 Lemmas and Corollaries

The techniques that we build up study the characteristics of a path with the use of packet-pair and packet-train probes.

These techniques are based on properties enounced in the followings Lemmas and related Corollaries. These are also exposed in [1].

Remember that most of the followings properties are valid as long as the assumptions stated in Section 2.2 are respected.

Lemma 1

Packet-Pair Property. In a path L composed by n physical links $L_1, L_2, \cdots, L_n$ with capacity bandwidths $b_1, b_2, \cdots, b_n$ respectively. If we inject a probe like $[pp]$ at L_1, with $D(p) = L_n$ then the interarrival time of these two packets at L_n is

$$\Delta_n = \frac{s(p)}{\underset{1 \leq k \leq n}{min}\ b_k} \tag{2.1}$$

Corollary 1.1 Let L be a path composed by n physical links $L_1, L_2, \cdots, L_n$ with capacity bandwidths $b_1, b_2, \cdots, b_n$ respectively. The bottleneck's capacity can be estimated through measurement of packet interarrival times and knowledge of packet size. In particular if the bottleneck is at link L_i, this can be calculated with the following formula:

$$b_i = \min_{1 \le k \le n} b_k = \frac{s(p)}{\Delta_n} \tag{2.2}$$

This property is used by almost every technique that estimates the bottleneck's capacity.

Lemma 2

Tailgating Property. Consider a path L of n physical links $L_1, L_2, \cdots, L_n$ with capacity bandwidths $b_1, b_2, \cdots, b_n$ respectively. If we inject a probe such as $[pq]$ at L_1, with $D(p) = D(q) = L_n$, and if $\left\{\forall k : 1 \le k \le n, \frac{s(p)}{s(q)} \ge \frac{b_{k+1}}{b_k}\right\}$, then the probe $[pq]$ will remain back-to-back over all links L_k.

$\left\{\forall k : 1 \le k \le n, \frac{s(p)}{s(q)} \ge \frac{b_{k+1}}{b_k}\right\}$ is equivalent to $\left\{\forall k : 1 \le k \le n, \frac{s(p)}{b_{k+1}} \ge \frac{s(q)}{b_k}\right\}$, therefore this means that, before the transmission of packet p at link $k+1$ is finished, the packet q is completely received at link k, for all the links in L.
This proves that the probe $[pq]$ will remain back-to-back along the path.

Lemma 3

Preservation of Spacing. Let L be a path of n physical links $L_1, L_2, \cdots, L_n$ with capacity bandwidths $b_1, b_2, \cdots, b_n$ respectively. If a probe of the form $[p][p]$ is injected at link L_1 with $D(p) = L_n$ and an interarrival time at link 1 of Δ^*, then Δ^* will be preserved along L if and only if $\frac{s(p)}{\Delta^*} \le \min_{1 \le k \le n} b_k$.

Proof: The transmission time of each probe packet, $[p]$, over the bottleneck link is $\frac{s(p)}{\min_{1 \le k \le n} b_k}$. Hence, if $\frac{s(p)}{\min_{1 \le k \le n} b_k} > \Delta^*$ then the interarrival time between the two packets at each link before the bottleneck link will remain Δ^* and from the following links this interval will be expanded to $\frac{s(p)}{\min_{1 \le k \le n} b_k}$ till the destination L_n.

Lemma 3 shows that in order to preserve an interarrival time (Δ_i) through a subpath $L_{i+1}, \cdots, L_n$, the condition $\frac{s(p)}{\Delta_i} \leq \min_{(i+1) \leq k \leq n} b_k$ must hold.

Lemma 4 In a path L of n physical links $L_1, L_2, \cdots, L_n$ with capacity bandwidths $b_1, b_2, \cdots, b_n$ respectively. If we inject a probe of the form $\left[p_j m \left\{p_j q_j\right\}^{r-1} p_j m\right]$ at link L_1 with $s\left(p_j\right) \geq s\left(m\right) = s\left(q_j\right)$, $D\left(p_j\right) = D\left(q_j\right) = L_j$ and $D\left(m\right) = L_n$, let Δ_j be the interarrival time between the m packets at the link L_j then

$$\Delta_j = \frac{r\left(s\left(p_j\right) + s\left(m\right)\right)}{b_{1,j}} \tag{2.3}$$

We refer to the m packet with the title of *marker packet* and to the p_j packet with the title of *magnifier packet*. The parameter r is called *Train Size*.

Corollary 4.1 Consider a path L of n physical links $L_1, L_2, \cdots, L_n$ with capacity bandwidths $b_1, b_2, \cdots, b_n$ respectively. If we inject a probe like $\left[p_j m \left\{p_j q_j\right\}^{r-1} p_j m\right]$ at link L_1 with $s\left(p_j\right) \geq s\left(m\right) = s\left(q_j\right)$, $D\left(p_j\right) = D\left(q_j\right) = L_j$ and $D\left(m\right) = L_n$, let Δ_j be the interarrival time between marker packets at the link L_j, then this Δ_j will be preserved over $L_{j+1}, \cdots, L_n$ if and only if

$$\frac{b_{1,j}}{b_{j+1,n}} \leq \frac{r\left(s\left(p_j\right) + s\left(m\right)\right)}{s\left(m\right)} \tag{2.4}$$

Corollary 4.1 can be obtained using Lemma 3 and Lemma 4, and shows the condition to be met to preserve the marker interarrival times from link L_j to link L_n.

Knowing Δ_j and the size of the marker and magnifier packets, if the Corollary 4.1 condition is valid, we can obtain $b_{1,j}$.

Observation 4.1.1: The value of r cannot be known without the knowledge of the link's properties in L. A solution to this problem can be found in Section 2.5.

Lemma 5 Let L be a sequence of n physical links $L_1, \cdots, L_i, \cdots, L_j, \cdots, L_n$ with capacity bandwidths $b_1, \cdots, b_i, \cdots, b_j, \cdots, b_n$ respectively. If we inject a probe of the form $\left[p_{i-1}m\left\{\{p_\theta q_\theta\}^{r-1}p_\theta m : i \le \theta \le j\right\}\right]$ with $s(p_\theta) \ge s(m) = s(q_\theta)$, $D(p_\theta) = D(q_\theta) = L_\theta$, $D(p_{i-1}) = L_{i-1}$, $D(m) = L_n$, if $b_{1,i-1} < b_{i,j}$ and
if $\frac{s(p_\omega)}{s(m)} < \frac{b_k}{b_{k-1}}$
then

$$\Delta_k = \frac{r(s(p_\omega) + s(m))}{b_{1,i-1}} \tag{2.5}$$

otherwise

$$\Delta_k = \frac{r(s(p_\omega) + s(m))}{b_{1,i-1}} + \frac{s(p_\omega)}{b_k} - \frac{s(m)}{b_{k-1}} \tag{2.6}$$

where $i - 1 \le \omega \le j$ and $i \le k \le j$.

We refer to L_i with the title of *initial egress link* and to L_j with the title *final egress link*.

Corollary 5.1 In a path L of n physical links $L_1, \cdots, L_i, \cdots, L_j, \cdots, L_n$ with capacity bandwidths $b_1, \cdots, b_i, \cdots, b_j, \cdots, b_n$ respectively. If we inject a probe of the form $\left[p_{i-1}m\left\{\{p_\theta q_\theta\}^{r-1}p_\theta m : i \le \theta \le j\right\}\right]$ with $s(p_\theta) \ge s(m) = s(q_\theta)$, $D(p_\theta) = D(q_\theta) = L_\theta$, $D(p_{i-1}) = L_{i-1}$, $D(m) = L_n$, if $\exists L_x \in [L_i, L_{i+1}, \cdots, L_j]$: the Tailgating Property is satisfied then

$$\Delta_{k^*} \triangleq \max_{i \le k \le j} \Delta_k$$

$$b_{i,j} = \frac{s(p)}{\Delta_{k^*} - \frac{r(s(p)+s(m))}{b_{1,i-1}} + \frac{s(m)}{b_{k^*-1}}} \tag{2.7}$$

so L_{k^*} is the bottleneck.

Proof: If the Tailgating Property is satisfied, $\frac{s(p)}{s(m)} \ge \frac{b_k}{b_{k-1}}$ then

$$\Delta_k = \frac{r(s(p) + s(m))}{b_{1,i-1}} + \frac{s(p)}{b_k} - \frac{s(m)}{b_{k-1}}$$

$$\Delta_{k^*} \triangleq \max_{i \le k \le j} \Delta_k = \frac{r(s(p)+s(m))}{b_{1,i-1}} + \frac{s(p)}{b_{k^*}} - \frac{s(m)}{b_{k^*-1}}$$

Hence, L_{k^*} is the bottleneck in $\left[L_i, \cdots, L_j\right]$, so the last thing to do is to get b_{k^*} from Equation (2.6) and then we obtain Equation (2.7).

Observation 5.1.1: Δ_k is the interarrival time of the $(\theta - i + 1)$-th marker packet and the $(\theta - i + 2)$-th marker packet at link L_k with $i \le \theta \le j$.
Therefore, Δ_k is the interarrival time to use to compute b_k.

Observation 5.1.2: Δ_{k^*} is formed by three components:

1. $\frac{r(s(p)+s(m))}{b_{1,i-1}}$, that is Δ_{i-1} (the dispersion time between each marker packet and its following at link L_{i-1}), plus
2. $\frac{s(p)}{b_{k^*}}$, that is the interval time spent between the receiving of the first and the last byte of the magnifier packet, at the end of link L_{k^*}, and less
3. $\frac{s(m)}{b_{k^*-1}}$, that is the interval time spent between the receiving of the first and the last byte of the marker packet, at the end of link L_{k^*-1}.

Hence, Δ_{k^*} is due to the dispersion time between a marker packet and its following at link L_{i-1} plus a certain interval.

Observation 5.1.3: If $\frac{s(p_\omega)}{s(m)} < \frac{b_k}{b_{k-1}}$ then the estimation of $b_{i,j}$ could not be accomplished.

Corollary 5.2 In a path L of n physical links $L_1, \cdots, L_i, \cdots, L_j, \cdots, L_n$ with capacity bandwidths $b_1, \cdots, b_i, \cdots, b_j, \cdots, b_n$ respectively. If we inject a probe of the form $\left[p_{i-1} m \left\{ \{p_\theta q_\theta\}^{r-1} p_\theta m : i \le \theta \le j \right\}\right]$ with $s(p_\theta) \ge s(m) = s(q_\theta)$, $D(p_\theta) = D(q_\theta) = L_\theta$, $D(p_{i-1}) = L_{i-1}$, $D(m) = L_n$, we can estimate $b_{i,j}$ if and only if the tailgating property is met and the following condition is valid

$$\frac{b_{1,i-1}}{b_{j+1,n}} \le \frac{r(s(p)+s(m))}{s(m)} \tag{2.8}$$

2.4 Prefix Path Estimation VS Target Subpath Estimation

We used packet dispersion techniques to infer information about the capacity of a segment in a path.

Generally these techniques are able to estimate the bottleneck's capacity in a given moment but they are not able to estimate the capacity of any other link in this path.

This is due to the fact that packet dispersion time in a given prefix path could not be preserved along the path.

This happens when the capacity of the following links is less than the capacity of the links we want to examine.

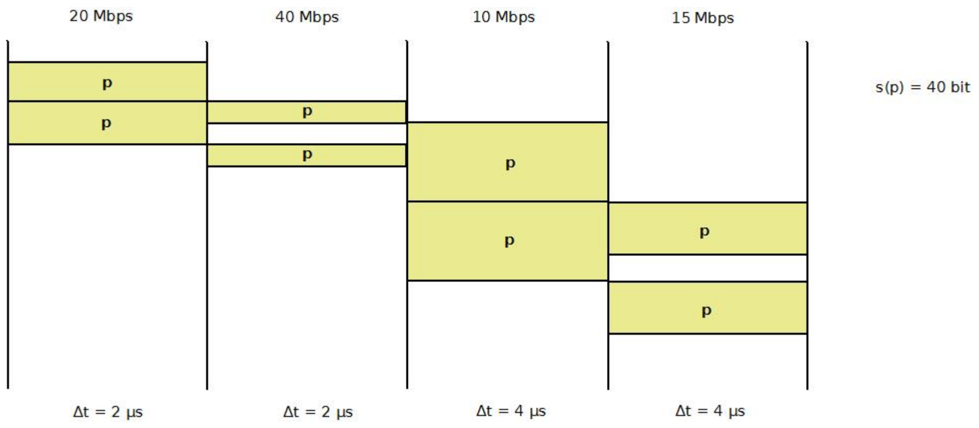


Figure 2.1: Bottleneck

In Figure 2.1 the packet dispersion time at the end of the path is bound only to the throughput's bottleneck.

A way to overcome this problem is to increase the time interval between the two marker.

This can be done sending other packets between the two marker, in this way the bottleneck link does not change the interarrival time of these packets.

Hence to estimate the throughput of the first segment in a path we should use a probe of this type $[p_j m \{p_j q_j\}^{r-1} p_j m]$. E.g. if we estimate $b_{1,3}$, with $r = 2$, then we will send the probe $[p_3, m, p_3, q_3, p_3, m]$, as shown in Figure 2.2.

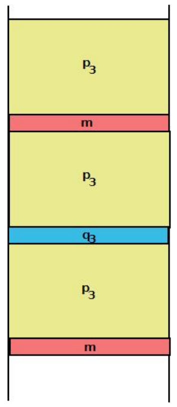


Figure 2.2: Simple probe for $b_{1,3}$ estimation

This packet-train probe is built in such a way that the condition enunciated in Lemma 3 is respected. In particular the choice of the train size (r) is fundamental.

In Section 2.5 will be presented a way to make a correct choice of this parameter.

The receiver, using Lemma 4 and knowing that the condition in Corollary 4.1 holds, can retrieve any throughput $b_{1,j}$.

It is interesting to note that this technique allow us to infer specifically b_1.

The technique presented till now is a packet dispersion technique and in this type of technique it is very important to know the instant when a packet arrives to the receiver. A mechanism that can invalidate this type of estimation is the Interrupt Coalescence.

To avoid flooding a host system with too many interrupts, packets are collected and one single interrupt is generated for multiple packets.

This mechanism is called *Interrupt Coalescence* (*IC*) and may affect the estimation of a packet dispersion technique because it can change the interarrival time between each packet. This is due to the fact that the application that performs the estimation is at the application layer, so do not take the time when the packets are really received but when the application is woke up by an interrupt.

The Figure 2.3 shows the behaviour of the system when receives several interrupts.

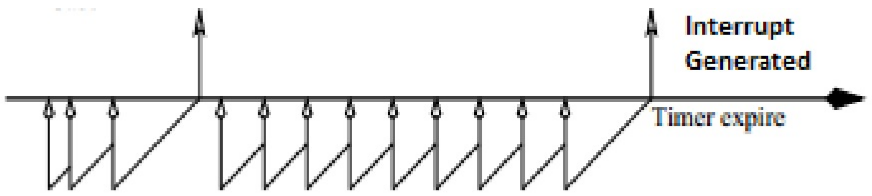


Figure 2.3: Interrupt Coalescence

In that probe the problem is that at the end point of the path arrived only two interrupts for the twelve marker packets received.

The solution is to transmit enough marker packets in order to trigger an interrupt. A solution based on the same considerations is also exposed in [7].

Hence, the probe used is of the form $\left[\left\{p_j m \left\{p_j q_j\right\}^{r-1}\right\}^{TL-1} p_j m\right]$, so the recipient receive *Train Length* (TL) marker packets.

E.g. if we want to estimate $b_{1,3}$, with $r = 2$ and $TL = 3$, we will send the probe $[p_3, m, p_3, q_3, p_3, m, p_3, q_3, p_3, m]$ like shown in Figure 2.4.

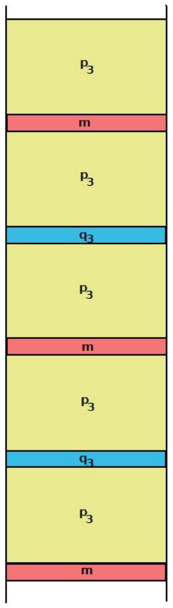


Figure 2.4: Complex probe for $b_{1,3}$ estimation

To estimate the prefix path bottleneck the formula becomes

$$b_{1,j} = \frac{r\,(s\,(p) + s\,(m))\,(TL-1)}{\Delta_j} \tag{2.9}$$

where Δ_j is the interarrival time between the last and the first marker packet in the probe.

Our other goal is to estimate b_i with $2 \leq i \leq n$, but before that we have to see in general how it is possible to estimate $b_{i,j}$ with $2 \leq i < j \leq n$.

We can observe that using the packet-train probe seen previously we can estimate $b_{1,i-1}$ and $b_{1,j}$ in a path L of n physical links $L_1, \cdots, L_i, \cdots, L_j, \cdots, L_n$ with capacity bandwidths $b_1, \cdots, b_i, \cdots, b_j, \cdots, b_n$ respectively. If $b_{1,i-1} > b_{1,j}$ then $b_{i,j} = b_{1,j}$, because this means that the bottleneck in the segment $\left[L_1, \cdots, L_i, \cdots, L_j\right]$ is placed between $\left[L_i, \cdots, L_j\right]$.

Otherwise, we have to use Corollary 5.1 in order to estimate $b_{i,j}$.

The estimation made using the probe $\left[p_{i-1}m\left\{\{p_\theta q_\theta\}^{r-1} p_\theta m : i \leq \theta \leq j\right\}\right]$ is affected by IC but, unlike the probe used in the estimation of $b_{1,j}$, we cannot send more than two marker packets to estimate a single link in the subpath.

The explanation of this statement lies in the different way we calculate $b_{1,j}$ and $b_{i,j}$ and, more importantly, the different properties exploited by these estimations.

In the prefix path estimation we can estimate $b_{1,j}$ simply using Lemma 4, to calculate the throughput, and Corollary 4.1, in order to guarantee that the dispersion time will be preserved.

In this case Δ_{k^*} depends on the packet dispersion time between a marker packet and its following at link L_i and the size of the packets in the probe.

The estimation of $b_{i,j}$ depends on the same elements of the prefix path estimation but, above all, depends on the transmission time of the magnifier packet at link L_{k^*} and the transmission time of the marker packet at link L_{k^*-1}. In fact Δ_{i-1} is expanded by this other components.

Therefore if we transmit a probe of this form $\left[p_{i-1}m\left\{\{p_\theta q_\theta\}^{r-1} p_\theta m : i \leq \theta \leq j\right\}^{TL-1}\right]$ then the estimation of b_θ depends only on the first and second marker for every subtrain of length $TL-1$.

In the following Figure is shown an example of the behaviour of this probe when $i = j$, $r = 1$ and $TL = 2$.

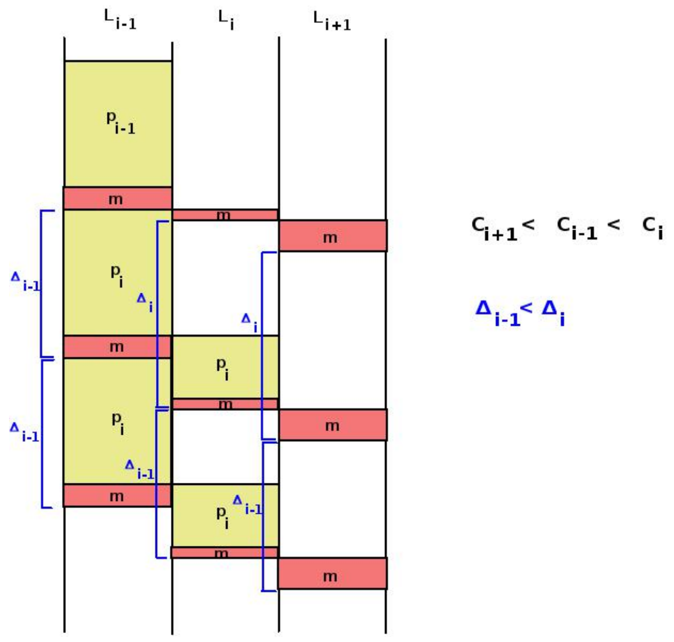


Figure 2.5: Behaviour of the target subpath Probe

In Figure 2.5 we can see that Δ_i can be obtained only between the first pair of marker packets.

The only thing to do is to perform many estimations of this parameter and to find the best among them.

At this point we can perform successfully an estimation of the prefix path and the target subpath under the assumptions made in Section 2.2.

The strongest assumption made is to consider a network free from cross traffic.

The cross traffic is the traffic composed by the packets that does not belong to our probing application and that pass through some of the links in our path.

The Figure 2.6 shows this scenario.

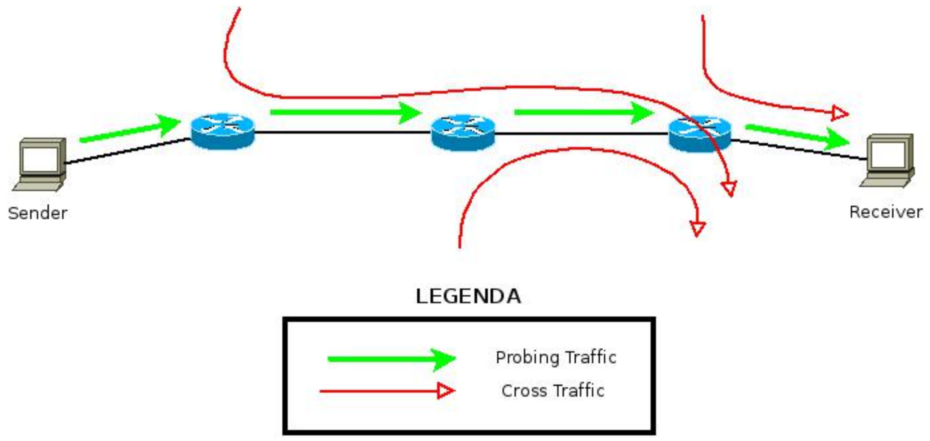


Figure 2.6: Cross traffic

The effect of the cross traffic on the probing traffic is either to compress or to expand the interval time between the packets.

This effect is due to the queue delays introduced on a router by the cross traffic.

The expansion of the interval time can happen when a router decide to send another packet between two packets belonging to the same probe.

Instead the compression of the interval time can occur when a router receives a probe packet but do not retransmit it immediately altering its interarrival time with the successive marker packet.

The Figure 2.7 shows an example of these effects produced by the cross traffic.

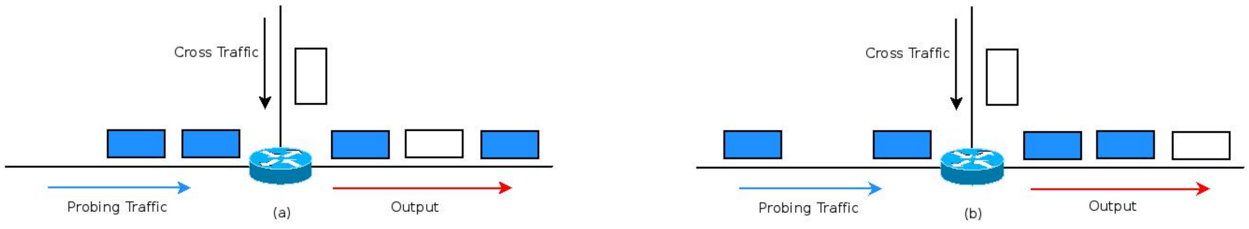


Figure 2.7: (a) Expansion of the interval time between the probing packets (b) Compression of the interval time between the probing packets

The assumption to consider the network free from cross traffic is not valid in the current Internet environment. Hence we have to make more accurate the estimation.

Obviously a method is to perform many estimations of the same parameter and get the most reliable among them.

The problem is how to understand if an estimation is reliable. This is a common problem for many estimation techniques.

We have decide to use a parameter called *Minimum Sum Delay*, introduced in [2].

The Minimum Sum Delay is computed as the sum of the smallest delay experienced by the first marker packet in a probe (t_f) and the smallest delay experienced by the last marker packet in a probe (t_l).

Once computed this parameter we will consider Δ_j as the difference between t_l and t_f.

This parameter has the quality to select the interval time that comes closet to what the marker packets would experience in a network free from cross traffic.

The Figure 2.8 shows an example of the computation of the Minimum Sum Delay and of the Δ_j.

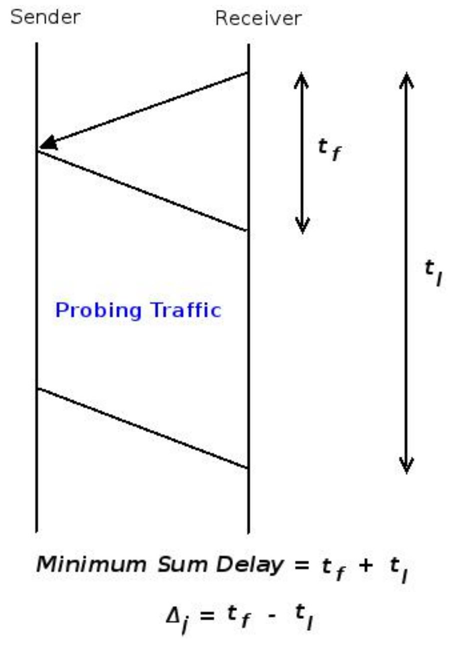


Figure 2.8: Minimum Sum Delay and Δ_j

2.5 Choice of train size

The target of this thesis is to be able, with a smartphone, to estimate the throughput of each link in the path that there is between this device and a server.

The estimation of the throughput of a specific link is the particular case of the estimation of a target subpath, where the parameter to be studied is b_i and $b_{i,i} = b_i$, with $1 \leq i \leq n$.

The algorithm to use in order to estimate b_i is the following:

Algorithm 2.1 Estimation of the capacity of each link in a path

```
Start the estimations using $r = 1$

1. Send $n$ prefix probe to the client to estimate $b_1$
2. Client calculates $b_1$
3. for every other link $L_i$ in the path
    (a) Send $n$ prefix probe to the client to estimate $b_{1,i-1}$
    (b) Client calculates $b_{1,i-1}$
    (c) Send $n$ prefix probe to the client to estimate $b_{1,i}$
    (d) Client calculates $b_{1,i}$
    (e) if $b_{1,i-1} > b_{1,i}$
    (f) then
        i. $b_i = b_{1,i}$
    (g) otherwise
        i. Send $n$ target probe to the client to estimate $b_i$
        ii. Client calculate $b_i$
    (h) end if
4. end for
```

This algorithm is effective as far as the choice of the value of r respects the constraints enounced in Corollary 2.3.

The problem is that the estimations are made without knowing the characteristics of the links in the path. This means that before the beginning of the estimations we cannot know the right value of the train size.

After several experiments we found that the estimations, made using $r = 1$ of prefix path and target subpath, have some typical behaviour. In particular the estimations of prefix path are underestimated and consequently the estimations of the target subpath are overestimated.

The train size is used to preserve the interarrival time between the marker packets at a certain link, in fact when we want to estimate $b_{1,j}$ if we do not use a proper train size then Δ_j will not be preserved and in particular it will be expanded due to the bottleneck link capacity, b_k. This explains why, in these cases, the estimations of the prefix path are underestimated. In particular we have that $\Delta_n = \Delta_k = \frac{s(m)(TL-1)}{b_k}$.

An important thing to highlight is the fact that only the links before the bottleneck link will be affected by the wrong choice of the train size, because Δ_j is the interarrival time of the marker packets at the bottleneck link in the subpath $\left[L_1, \cdots, L_j\right]$ and if L_k, the bottleneck link of the entire path, is in $\left[L_1, \cdots, L_j\right]$ then $\Delta_k = \Delta_j$ and so this time cannot be expanded in the following links.

The underestimation of the prefix path $b_{1,i-1}$ also affects the estimation of the target subpath $b_{i,j}$ because this estimation make use of $b_{1,i-1}$.

Recalling the Formula 2.7 we can see that the estimation of $b_{i,j}$ is based on some fixed elements, such as the packets size and the train size, on some previous estimations, $b_{1,i-1}$ and b_{k^*-1}(and on the parameter calculated in this step, Δ_{k^*}.

$$b_{i,j} = \frac{s(p)}{\Delta_{k^*} - \frac{r(s(p)+s(m))}{b_{1,i-1}} + \frac{s(m)}{b_{k^*-1}}} \tag{2.7}$$

If the train size is not of the right value then $b_{1,i-1}$ is been underestimated and b_{k^*-1} is been overestimated. These parameters contribute to overestimate $b_{i,j}$.

After seeing the consequences of an incorrect choice of the train size we have to find a way to realizing this. A solution is to perform a first estimation of all the links in the path and at a later stage to analyze the estimations of the links before the bottleneck link.

On the basis of where is placed the bottleneck link we have to keep different behaviour. In particular there are three different cases:

1. the bottleneck link is the first link then we can be confident that the interarrival time between the marker packets will be preserved along the path for every link, so is not needed any other estimation

2. in case the bottleneck is placed at link L_2 then we cannot know if the estimation is been underestimated, so we have to perform another estimation using a large value for the train size in order to avoid any estimation errors

3. in the last case, where the bottleneck link is in $[L_3, \cdots, L_n]$, we have to check if the estimation can be considerate valid. If there is a b_i much larger than b_k, we can assume that there could be an estimation error, so we should perform another estimation with a bigger value for the train size. If the estimation was correct then it should not change too much.

From these considerations we have improved the algorithm in order to perform a correct estimation even in those cases. The result is shown in the following section.

It is important to remark the fact that the estimation of a target subpath can be done only if the condition expressed in Observation 5.1.3 is respected.

2.6 Improved algorithm

Taking into account all the considerations made until now, we can build an algorithm able to estimate the throughput of each link in the path that there is between this device and a server.

We have used a parameter called *MAX_R* to represent an upperbound for the train size. Is been set an upperbound to the train size because otherwise it could be no end to the estimations.

The value of *MAX_R* is set to the initial value of the train size plus 4 because in a typical real environment the bottleneck is placed in the first links and, in these links, they do not have capacities much different among them.

The point 5 of the algorithm is the improvement described in Section 2.5.

Algorithm 2.2 Estimation of the capacity of each link in a path (improved)

```
Start the estimations using $r = 1$
1. Send $n$ prefix probe to the client to estimate $b_1$
2. Receiver calculates $b_1$
3. **for** every other link $L_i$ in the path
   (a) Send $n$ prefix probe to the client to estimate $b_{1,i-1}$
   (b) Receiver calculates $b_{1,i-1}$
   (c) Send $n$ prefix probe to the client to estimate $b_{1,i}$
   (d) Receiver calculates $b_{1,i}$
   (e) **if** $b_{1,i-1} > b_{1,i}$
   (f) **then**
       i. $b_i = b_{1,i}$
   (g) **otherwise**
       i. Send $n$ target probe to the client to estimate $b_i$
       ii. Receiver calculates $b_i$
   (h) **end if**
4. **end for**
5. Considering $L_k$ as the bottleneck link in the path
   (a) **if** $k = 2$
   (b) **then**
       i. Repeat the estimations of $b_1$ using $r = MAX\_R$
       ii. **break**
   (c) **otherwise**
       i. **for** all link $L_i$ in $[L_1, \cdots, L_{k-1}]$ **while** $r \leq MAX\_R$
          A. **if** $30 \cdot b_k \leq b_i$
          B. **then**
          C. Repeat the estimations of $[b_1, \cdots, b_{k-1}]$ using $r = r + 1$
          D. **break**
          E. **end if**
       ii. **end for**
   (d) **end if**
```

Chapter 3

Implementation

In the previous chapters we discuss the algorithm used in this project but in this chapter we provide a fully detailed description of the components of the system, both client and server components, and the communication protocol used between client and server.

This implementation has been project to be able to perform an estimation of both download and upload.

The components on the client side are conceived to be performed on a smartphone android, instead the server's components are developed for a classic host machine.

The server side of the system has been designed to be a service that a client can require to estimate the capacity of a single parameter $b_{i,j}$ with $1 \leq i \leq j \leq n$. This choice gives the greatest flexibility to the server to handle the connections of the clients and they can build autonomously their own estimations.

In the first part of this chapter we describe the single components and their functionalities, while in the second part we illustrate in detail how these components interact each other.

3.1 System components overview

The server side of the system is formed by the following components:

1. CapacityServer
2. ServerTask
3. Scheduler

CapacityServer is a simple component that is designated to receive the request that the clients submits to the server.

ServerTask is a module that handles the communication with a single client. This module must negotiate the parameters of the estimation and has to receive/send the probing traffic.

Eventually the Scheduler is the component that rules the scheduling of the requests. A ServerTask must receive the permission from the Scheduler in order to start the estimation with the client.

Instead the main components on the client side of the system are:

1. MainActivity
2. MyCoordinator
3. PathEstimator
4. CapacityEstimation

MainActivity provide a simple interface for the user to interact with the application.

The MyCoordinator component is an Android Service that runs in background and periodically starts the estimation of each link's capacity in the path that there is between the smartphone and the server machine.

The PathEstimator component repeatedly calls the CapacityEstimation in order to estimate the capacity of all the link in the path. According to the algorithm, it performs other estimations of some links even if it is not clear that the valuation for these links are reliable. Moreover it saves the results obtained in a log file to allow their consultation to the client.

CapacityEstimation is the module designated to perform an estimation of the capacity of a single link. To perform this estimation it could be necessary to estimate a few parameters for this link like explained in the Chapter 2.

In substance PathEstimator handle the operations to perform the estimations and CapacityEstimation actually estimates the capacity of a certain link.

There are also some components that are common to both the sides. These components are the following:

1. CapacityReceiver

2. CapacitySender

3. TrainBuilder

CapacityReceiver is called by CapacityEstimation and ServerTask when is necessary to receive the probing traffic.

The CapacitySender is called by the same components but it is used to send the probing traffic. In reality this component uses TrainBuilder to send the packets and it coordinates the sending of the packets.

TrainBuilder is the module that sends the probes described in Chapter 2.

In the Section 3.2 we will provide a fully description of the modules used on the server side, instead in Section 3.3 we will describe the components of the client side.

Eventually in Section 3.4 is provided a comprehensive description of the modules common to both the sides.

In Figure 3.1 is shown the system's architecture.

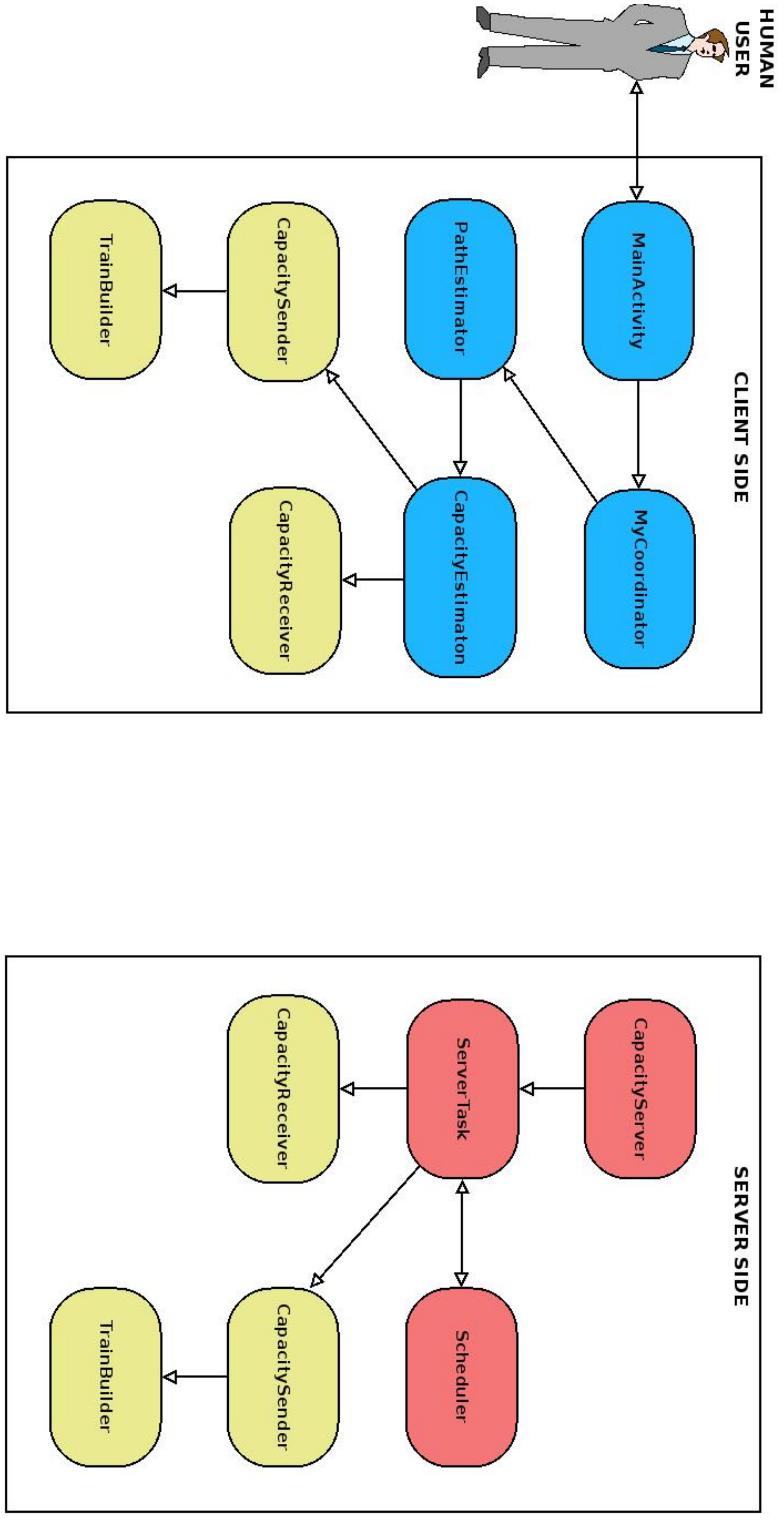


Figure 3.1: System's architecture

3.2 Server side

In this section the components used only by the server will be illustrated. In particular are reported only the components only used by the server side.

The other components that used both server and client side are presented in Section 3.4.

In Figure 3.2 is shown the architecture of the server side.

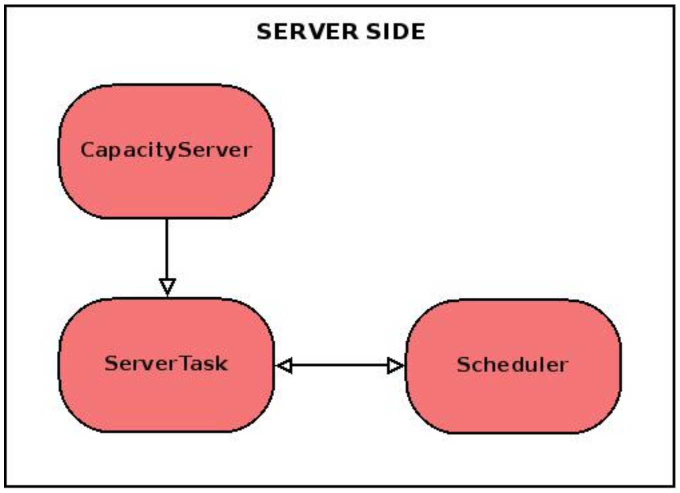


Figure 3.2: Server's architecture

3.2.1 CapacityServer

The CapacityServer is a Java thread appointed to wait for some TCP requests by the clients and assigns to everyone a specific ServerTask to handle the estimation.

This choice allows the server to remain in a listening mode also during the estimation phase, it will be the ServerTask component that will check the correctness of the request and, if so, it will execute the estimation.

Remember that the estimation performed by the server consist of only the evaluation of $b_{i,j}$ where $1 \leq i \leq j \leq n$.

3.2.2 ServerTask

ServerTask is a Java thread that handle the communication with the client, in particular its execution can be divided in two phases:

1. setup phase
2. estimation phase

In the setup phase this module receives the parameters of the estimation requested by the client, registers this request on the Scheduler and gives a response to the client.

The request must contain the following elements:

- $s(m)$: the size of the marker packet
- $s(p)$: the size of the magnifier packet
- r: the train size
- L_i: the initial egress link
- L_j: the final egress link
- TL: the train length
- $b_{1,i-1}$: if the estimation is of a specific link L_i, where $i \neq 1$, then this is the capacity of the prefix path $[L_1, \cdots, L_{i-1}]$, otherwise this parameter must be set to -1 because it is not significant
- *mode*: the modality of the estimation, either download or upload
- b_{i-1}: if the estimation is of a specific link L_i, where $i \neq 1$, then this is the capacity of the previous link L_{i-1}, otherwise this parameter must be set to -1 because it is not significant
- *phoneID*: the identificator of the smartphone
- *clientVersion*: the version of client's software

Hence, ServerTask checks the correctness of the request and tries to register it on the Scheduler.

If the Scheduler notices that there is a long queue of requests that are hanging on then notifies to the ServerTask that this request cannot be served at this moment.

The response can be either a LONG_QUEUE message if there are too many pending requests or an ACK message otherwise. In the first case execution is terminated otherwise we proceed to the estimation phase.

The estimation phase is where the estimation is really performed. The client can request an estimation in either download or upload.

When the client requests an estimation in download the ServerTask calls the CapacitySender component to send the probing traffic, instead when the estimation is in upload then it calls the CapacityReceiver component to receive the probing traffic.

When this component has finished its execution, ServerTask points the fulfillment of the request to the Scheduler and write in a log file the results of the estimation, so these estimations can be checked in the future.

In case there are some network problems and ServerTask cannot complete the estimation before it ends its execution it must remove its request from the scheduler's queue to allow the execution of the other pending requests.

3.2.3 Scheduler

The Scheduler is a Java object that schedules the requests that must be served. In particular it implements a *FIFO* (*First In First Out*) list of requests (queue of requests), where the length of the list cannot exceed a certain threshold.

When the queue is full it does not accept any other request.

The request that is in top of the list is the request to be served by the instance of ServerTask assigned to it.

Scheduler provides the following methods:

- addRequest
- removeRequest
- getTurn

The method addRequest checks if the queue of the requests is not full and, if so, it adds in tail the request. When the queue is full it notifies to ServerTask that the request cannot be accomplished and do not add it to the queue.

RemoveRequest is invoked when ServerTask wants to delete a request that is been served. Hence this method delete the first element of the queue.

When ServerTask has successfully added its request, it means that this request is in the queue of the Scheduler but we do not know in what position is placed. This request will be served but not necessary as first.

The method getTurn returns only when the request that can be served is the request assigned to this instance of ServerTask. In other words this method returns only when this request has become the first request in the queue.

Hence, the method getTurn must be invoked only after calling the method addRequest and it must gave a positive result.

In Figure 3.3 is shown the outline of the scheduler described here.

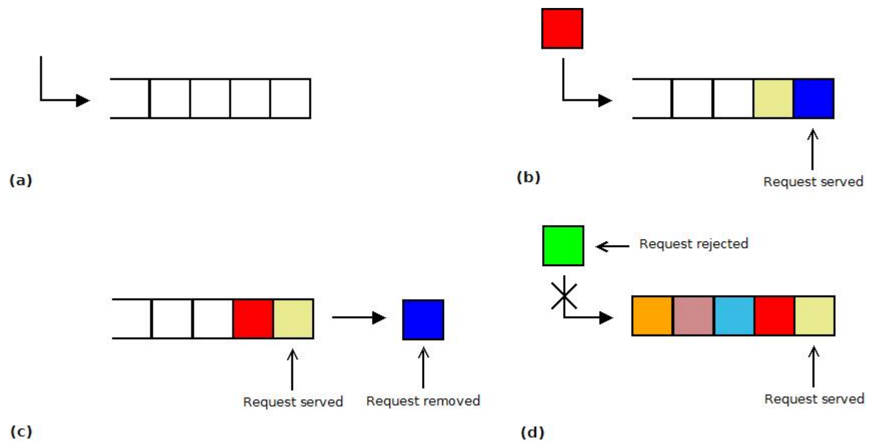


Figure 3.3: (a) Scheduler's queue empty (b) Add request to the Scheduler's queue (c) remove request from the Scheduler's queue (d) Scheduler's queue full therefore the request is rejected

An important thing to highlight is the fact that all these operations must be executed in mutual exclusion.

On the server there is a single instance of the Scheduler that is passed to every instance of ServerTask therefore Scheduler is a shared resource.

To ensure that the scheduler's queue remains in a consistent state it is important that these operations are executed in mutual exclusion.

3.3 Client side

In this section will be described the components that are client-specific without those modules that are common to both the sides.

In Figure 3.4 is shown the architecture of the client application.

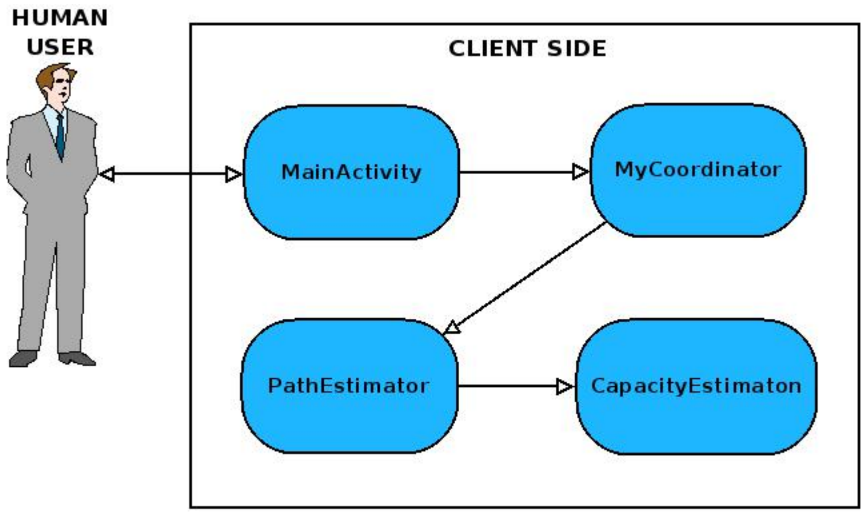


Figure 3.4: Client’s architecture

3.3.1 MainActivity

MainActivity is the front-end of this Android application. In particular this is an Android Activity.

An Android Activity is an application component that provides an interface towards the user.

The user can use this component to perform the following functionalities:

1. starts the process of estimation
2. stop the process of estimation
3. shows the results obtained until that moment

4. deletes the results saved

The first functionality refers to the possibility to start MyCoordinator to begin a campaign of estimations of the link's capacity, instead the second functionality stops this campaign of estimations.

If the user stops the campaign during an estimation in that case the system finishes to estimate the capacities of the remaining links in the path both download and upload and only when it completes this set of estimations it stops the campaign.

The other two functionalities allow the user to see the estimations made until that moment and allow him to delete it in the case these are no more interesting.

The following figure shows the interface provided to the user.

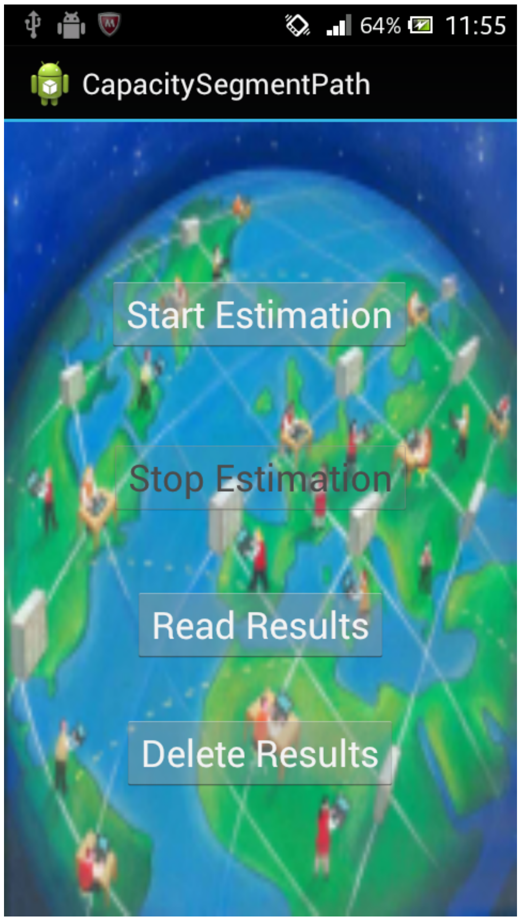


Figure 3.5: MainActivity

3.3.2 MyCoordinator

MyCoordinator is an extension to an Android Service. An Android Service is an application component that can perform long-running operations in the background of the main application that has called it.

In particular MyCoordinator must periodically call PathEstimator to estimate the link's capacities along the path both download and upload. This set of estimation is called campaign.

We used a Service because this component must run without affect the performance of MainActivity, in fact it is in PathEstimator that the estimation is really performed. In this way the interaction of MainActvity with the user remains fluid.

3.3.3 PathEstimator

PathEstimtor is a Java thread that follows the Algorithm 2.2 to estimate the link's capacity in both download and upload. It follows the steps of the algorithm calling CapacityEstimation to estimate a single capacity.

When it finishes to estimate the capacities of all the links in the path it checks if the estimations are made with a correct value of train size. Whenever it realizes that the value of train size was not correct it restarts the estimations for the interested links.

Before the end of the execution this module saves the results into a log file in order to allow the user to read these estimations.

3.3.4 CapacityEstimation

The estimation of the capacity of a certain link is accomplished by the module CapacityEstimation.

This module is a Java thread able to estimate the capacity of a link in both download and upload.

According to the algorithm, to estimate the capacity of a link in the path either download or upload it performs some estimations of either prefix path or target subpath.

When it performs a prefix path estimation it sets the value of the train length depending on the type of network to which the smartphone is connected.

The value of the train length is chosen accordingly to what explained in [2].

The next step is to send the request of the desired parameter to the server and wait for the response.

It can receive either a LONG_QUEUE or an ACK message. When it receives a LONG_QUEUE message it stops the estimation instead when it receives an ACK message it continues to the real estimation phase.

The estimation in download is performed in two steps:

1. receive the probing traffic
2. analyze the results

To receive the probing traffic CapacityEstimation calls a specific component responsible for carry out this task. The component called is CapacityReceiver.

When CapacityReceiver ends, CapacityEstimation collects the results retrieved by it and calculates both the Minimum Sum Delay and the throughput of the parameter estimated.

CapacityEstimation repeats these operations to estimate the parameters needed to esteem the capacity of the link desired.

Before the end of the execution this module saves the results into a log file in order to allow the user to read these estimations and sends the results to the server.

The estimation in upload is performed in other two steps:

1. send the probing traffic
2. receive the results

To send the probing traffic CapacityEstimation uses another component called CapacitySender. When CapacitySender ends its execution it waits for the server to analyze the probing traffic and then receives the results.

Eventually it saves the results in a log file to allow the user to read them.

3.4 Common components

In this section we describe the components that receive or send the probing traffic. For this reason these components are used by both the sides.

In the next subsections we described their functionalities.

3.4.1 CapacityReceiver

CapacityReceiver is a Java thread used to receive the probing traffic. At the beginning if it is used by the server it receives an UDP message called openPathMessage.

When a host is connected to a private network it cannot communicate with another host in a public network and vice versa.

To allow this host to communicate even in this case there is usually a border router in this network that is connected to a public network and it performs *NAT* (*Network Address Translation*).

The operation of NAT consists of the translation of the source address into another address of the router. This operation is usually performed when the source IP is a private address because otherwise it cannot communicate with other public networks.

In particular there are two types of NAT:

1. one-to-one NAT
2. PAT

In the first type there is a one-to-one mapping between the IP address of the source and an IP address of the router. The IP address of the router is a public address.

In *PAT* (*Port Address Translation*) it is changed both the source IP address and the source port. In this way many private address can be mapped in one public address.

The problem is that in upload the server cannot send the start packet (an UDP packet) to the client because it is behind NAT. In fact the client until that moment sent only TCP packets so the NAT made the mapping only for the TCP port.

The message called openPathMessage is used to create a mapping on the router with the UDP port on the router.

In this way when the server receives this UDP packet it will know what is the combination <IP, port> where the client can be reached.

When the estimation is performed in download this mechanism is inherently triggered by the start packet.

The next step is to start the real protocol to exchange the probing traffic. CapacityReceiver send an UDP message where it says to the server the number of the experiment to which has arrived and the value of train length to use. This UDP packet is called start packet.

At a later stage it receives the packet-train probe and check if the packets are correctly received. In case there is at least three experiments incorrect then it reduces the value of train length and restarts the estimation from the beginning. In this way the value of the train length is obtained in an adaptive manner.

When the packets are received CapacityReceiver saves the timestamps of each packet of every experiment. these information are given back to CapacityEstimation when it wants the results of the estimation.

3.4.2 CapacitySender

CapacitySender is a Java thread used to handle the sending of the probing traffic. This object in particular know only the protocol to use in order to communicate with the client side but do not deal with the construction of the packet-train probe.

When CapacitySender must communicate to the client with some UDP packets it takes advantage of the component TrainBuilder.

In case it is used on the client side it also calls TrainBuilder to send an UDP message called openPathMessage if so, the estimation is performed in upload.

As stated in the previous subsection this message is used to overcome the problems due to the mechanism of NAT and has no other purpose.

After this step, Similarly to what we see in CapacityReceiver, it calls TrainBuilder to receive an UDP packet to trigger the sending of the packet-train probe.

In the start packet is contained the value of train length to use and the experiment that the client waits for.

3.4.3 TrainBuilder

TrainBuilder is a Java object able to receive UDP packets and able to send UDP packets with a certain TTL. This object provides some useful methods to CapacitySender to communicate with the client application.

In particular it provides the following methods:

- createSocket
- receivePacket

- sendPacketWithMyTTL
- closeSocket
- sendOpenPath
- sendTrain

The methods createSocket, receivePacket, sendPacketWithMyTTL and closeSocket are been implemented using a native-code language such as C.

This is because the TTL parameter cannot be set in a Java environment, so we have to use a different language.

In Android there is a toolset called NDK that allows this operation. NDK is very similar to the framework JNI used in Java to recall the native code.

In this way we can use sendPacketWithMyTTL to send an UDP packet with a certain TTL, but consequently even the other methods mentioned before, that use the sockets, must be developed in native code.

In particular createSocket obtains a socket from the operating system and saves its identifier in an attribute of the TrainBuilder object. ReceivePacket receives an UDP packet from this socket and returns the payload of this packet. In the end closeSocket using the identifier of the socket closes the file descriptor associated, so that it can no longer refers to any file and may be reused.

The remaining methods make use of these functions to accomplish their own task.

The method sendOpenPath sends on the socket previously opened the openPathMessage to the client application. SendTrain builds a proper packet-train probe to estimate the parameter selected and sends these packets to CapacityReceiver.

3.5 Examples of execution

In this section is shown the interaction between entities involved in the system. Firstly, we describe how can be triggered the estimations of the capacities in the path both download and upload. Secondly, we provide a description of the execution flows interactions when a single estimation is performed in download. Thirdly, we show how the interactions change when the estimation is in upload.

Figure 3.6 shows how can be triggered a set of estimations and Listing 3.1 shows the pseudocode for this operation.

```
 1 ...
 2 public class MainActivity : Activity {
 3     public MyCoordinator coordinator;
 4     ...
 5     public void onPushStartButton() {
 6         coordinator.startService();
 7     }
 8     ...
 9 }
10 
11 public class MyCoordinator : Service {
12     public static final int TIMEOUT = 36000;     // in
           sec
13     ...
14     public void startService() {
15         while(true) {
16             PathEstimator pe = new PathEstimator();
17             pe.startEstimations();
18             int tStart = gettimeofday();
19             int tRestart = tStart + TIMEOUT;
20             wait(tRestart);     // set timeout
21         }
22     }
23 }
```

Listing 3.1: Start set of estimations

These estimations can be triggered in two ways:

1. the human user pushing the start button of MainActivity starts MyCoordinator to perform a campaign of estimations [lines 5-7]

2. the timeout expires and MyCoordinator starts a new set of estimations of the capacities in the path [line 20]

When MyCoordinator calls PathEstimator it restart the timeout and wait for the time when this expires. [lines 16-20]

In Figure 3.7 is shown an example of estimation in download. In Listing 3.2 is shown the pseudocode relative to this example on client wheras in Listing 3.3 is shown on server side.

```java
...
public class PathEstimator {
    ...
    public void startEstimations() {
        ...
        CapacityEstimation ce = new CapacityEstimation();
        double b1 = ce.calculateCapacity( 1, "download" );
        ...
    }
    ...
}

public class CapacityEstimation {
    ...
    public double calculateCapacity( int link, char mode[] ) {
        ...
        sendRequest();
        if( receiveResponse() != ACK )
            return SERVER_BUSY;
        CapacityReceiver cr = new CapacityReceiver();
        cr.start();
        Result res = cr.getResults();
        double capacity = res.getCapacity();
        sendResults( res );
        saveResults( res );
        return capacity;
    }
    ...
}

public class CapacityReceiver {
    private Result r;
    ...
    public void start() {
        ...
        sendStart();
        receiveProbingTraffic( r );
    }

    public Result getResults() {
        return r;
    }
    ...
}
...
```

Listing 3.2: Estimation in download of b1 on client side

```
...
public class CapacityServer {
    ...
    public void start() {
        Scheduler s = new Scheduler();
        ...
        while(1) {
            Request req = receiveRequest();
            ServerTask st = new ServerTask( req );
            st.start( s );
        }
    }
    ...
}

public class Scheduler {
    ...
    public boolean addRequest( Request req) {
        ...     // add the request to the queue
    }

    public void getTurn( req ) {
        ...     // wait for the turn of the
                // request req
    }

    public void removeRequest() {
        ...     // remove the request on the top
                // of the queue
    }
    ...
}

public class ServerTask {
    ...
    public void start( Scheduler s ) {
        ...
38          if( !s.addRequest( req ) ) {
39              sendResponse( LONG_QUEUE )
40          }
41          s.getTurn( req );
42          sendResponse( ACK );
43          CapacitySender cs = new CapacitySender();
44          cs.start();
45          Result res = receiveResults();
46          saveResults( res );
47          s.removeRequest();
48      }
49      ...
50  }
51
52  public class CapacitySender {
53      ...
54      public void start() {
55          TrainBuilder tb = new TrainBuilder();
56          ...
57          if( isClient )
58              sendOpenPath();
59          receiveStart();
60          tb.sendTrain();
61          ...
62      }
63      ...
64  }
65  ...
```

Listing 3.3: Estimation in download of b1 on server side

At the lines 6-7 on the client side PathEstimator calls CapacityEstimation to esteem b_1 in download. CapacityEstimation starts the estimation sending a request to CapacityServer [line 17 on client side] that, receiving this message, creates a new instance of ServerTask [lines 8-10 on server side].

At this point ServerTask must register its request to the Scheduler therefore it calls the method addRequest of the Scheduler [line 38 on server side]. In this case the Scheduler's queue is not full so ServerTask can wait for its turn with the method getTurn of Scheduler [line 41 on server side].

When this method will return, ServerTask sends an ACK message in the response TCP sent to the client and starts CapacitySender and waits for its completion [lines 42-44 on server side].

On the client side when CapacityEstimation receives the ACK message it starts CapacityReceiver to receive the probing traffic [lines 18-21 on client side].

CapacityReceiver sends a start message to CapacitySender to trigger the sending of the probing traffic [lines 36-37 on client side]. When CapacitySender finishes the sending of the probing traffic it ends its execution.

ServerTask resumes its execution and waits for the response from the client where is reported the capacity b_1 [line 45 on server side].

At lines 22-23 on client side CapacityReceiver ends its execution and CapacityEstimation retrieves the results obtained during the estimation and calculates the capacity b_1.

When CapacityEstimation finishes the calculation of b_1 it sends this parameter to ServerTask and after saving the result ends its execution [lines 24-26 on client side].

Eventually ServerTask receives and saves the result and removes its request from the Scheduler's queue with the method removeRequest [lines 45-47on server side].

In this way PathEstimator estimates b_1 in download.

In Figure 3.8 is shown an example of the estimation in upload. In Listing 3.4 and in Listing 3.5 is shown the pseudocode on client and server side respectively.

```
...
public class PathEstimator {
    ...
    public void startEstimations() {
        ...
        CapacityEstimation ce = new CapacityEstimation();
        double b1 = ce.calculateCapacity( 1, "upload" );
        ...
    }
    ...
}

```

```
public class CapacityEstimation {
    ...
    public double calculateCapacity( int link, char
        mode[] ) {
        ...
        sendRequest();
        if( receiveResponse() != ACK )
            return SERVER_BUSY;
        CapacitySender cs = new CapacitySender();
        cs.start();
        Result res = receiveResults();
        saveResults( res );
        return res.getCapacity();
    }
    ...
}

public class CapacitySender {
    ...
    public void start() {
        TrainBuilder tb = new TrainBuilder();
        ...
        if( isClient )
            sendOpenPath();
        receiveStart();
        tb.sendTrain();
        ...
    }
    ...
}
...
```

Listing 3.4: Estimation in upload of b1 on client side

```
...
public class CapacityServer {
    ...
    public void start() {
        Scheduler s = new Scheduler();
        ...
        while(1) {
            Request req = receiveRequest();
            ServerTask st = new ServerTask( req );
            st.start( s );
        }
    }
    ...
}

public class Scheduler {
    ...
    public boolean addRequest( Request req) {
        ...     // add the request to the queue
    }

    public void getTurn( req ) {
        ...     // wait for the turn of the
                // request req
    }

    public void removeRequest() {
        ...     // remove the request on the top
                // of the queue
    }
    ...
}


public class ServerTask {
    ...
    public void start( Scheduler s ) {
        ...
        if( !s.addRequest( req ) ) {
            sendResponse( LONG_QUEUE )
        }
        s.getTurn( req );
        sendResponse( ACK );
        CapacityReceiver cr = new CapacityReceiver();
        cr.start();
        Result res = cr.getResults();
        double capacity = res.getCapacity();
        sendResults( res );
        saveResults( res );
        s.removeRequest();
    }
    ...
}

public class CapacityReceiver {
    private Result r;
    ...
    public void start() {
        ...
        if( isServer )
            receiveOpenPath();
        sendStart();
        receiveProbingTraffic( r );
    }

    public Result getResults() {
        return r;
    }
    ...
}
...
```

Listing 3.5: Estimation in upload of b1 on server side

The execution of this estimation is similar to the execution of the estimation in download.

In this case the differences are that when is the turn of ServerTask, it starts Capacity Receiver and not CapacitySender [lines 44-45 on server side], and CapacityEstimation starts CapacitySender and not CapacityReceiver [lines 20-21 on client side].

The estimation in upload also involves the sending of the openPathMessage by CapacitySender [lines 34-35 on client side] and consequently the receiving of this packet by CapacityReceiver [lines 60-61 on server side].

Finally the last difference is that when CapacitySender and CapacityReceiver end their task, ServerTask get the results from CapacityReceiver and calculates b_1 [lines 46-47 on server side]. When it ends this estimation it sends this parameter to CapacityEstimation and removes the request from the Scheduler's queue [lines 48-50 on server side].

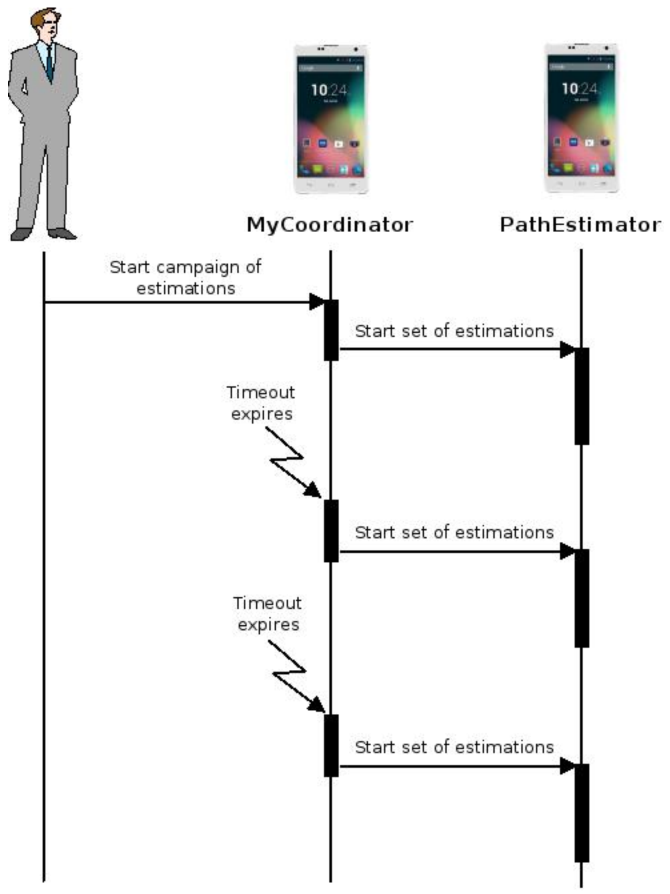


Figure 3.6: Starts of a campaign of estimations

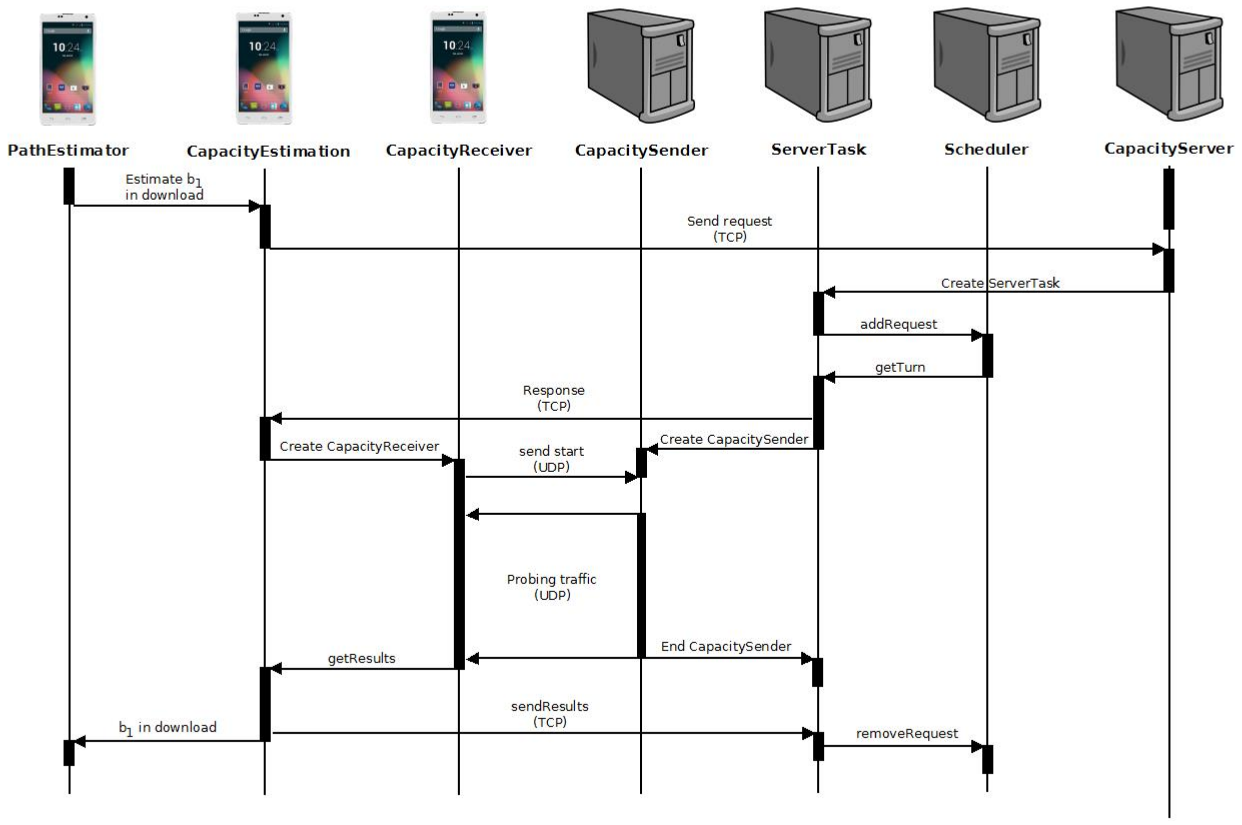


Figure 3.7: Estimation in download of b_1

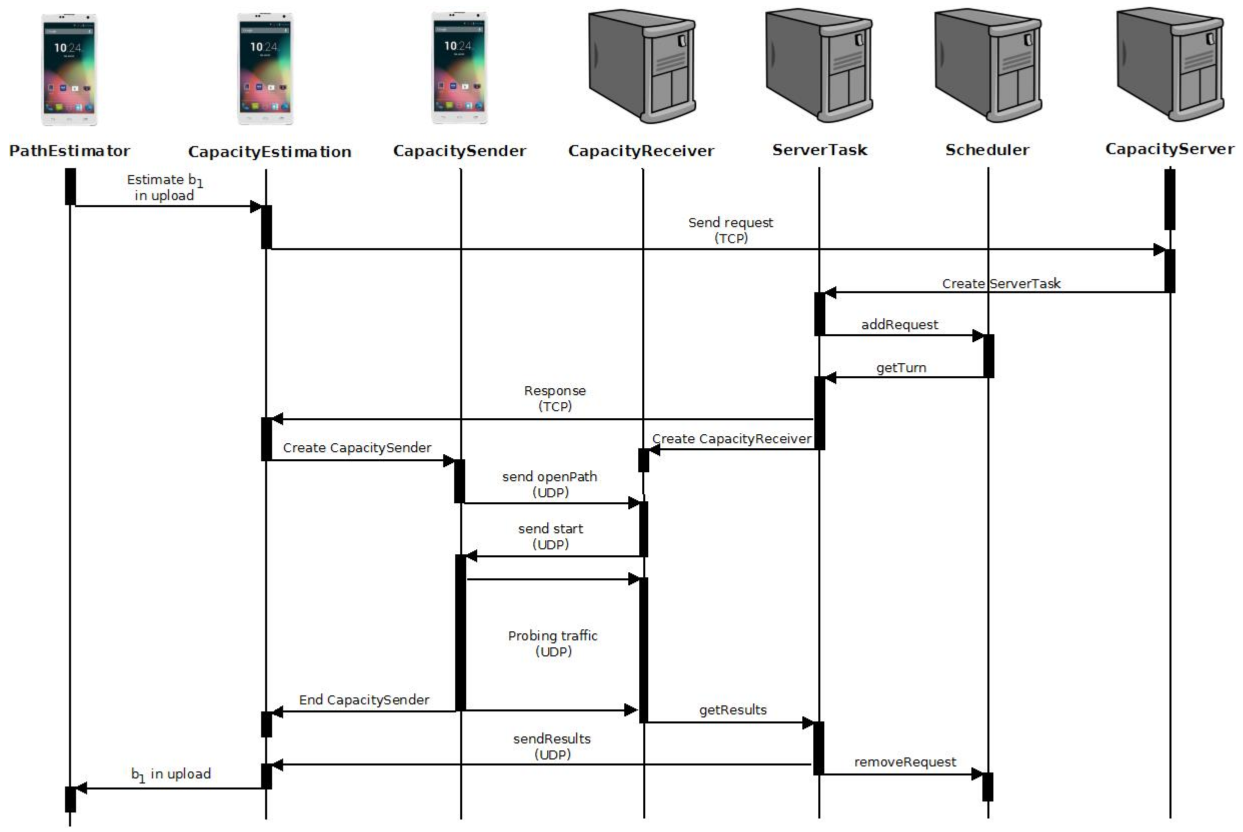


Figure 3.8: Estimation in upload of b_1

Chapter 4

Simulator NS-3

NS-3 (*Network Simulator 3*) is a free network simulator for Internet systems, released under the GNU GPLv2 license, that is public available for research, development and use.

The project NS-3 aims to develop an open simulation environment able to respond to the need to manage the entire simulation workflow, from simulation configuration to trace collection and analysis.

Furthermore it encourages the community contribution, peer review and validation of the software. The users can develop simulation models which are sufficiently realistic to allow NS-3 to:

- be used as a realtime network emulator
- interconnect with the real world
- reuse many existing real-world protocol implementations

4.1 Simulation models

The NS-3 core supports research for both IP and non-IP based networks. On the other hand the users typically prefer to study either wireless IP simulations, which involve models for Wi-Fi, WiMax (Worldwide Interoperability for Microwave Access) and LTE (Long Term Evolution), or a variety of static/dynamic routing protocols, such as OLSR (Optimized Link State Routing Protocol) and AODV (Ad-hoc On Demand Distance Vector) for IP-based applications.

Another facility offered by the NS-3 is a real-time scheduler that facilitates a number of "simulation-in-the-loop" use cases for interacting with real systems.

In particular the user can send or receive NS-3-generated packets on real network devices, and NS-3 can serve as an interconnection framework between virtual machines emulating the behaviour of the links in this virtual network.

This simulator can also serve to reuse real application and kernel code.

Frameworks for this purpose are developed, tested and evaluated for a Linux environment.

4.2 Key technologies

NS-3 is a C++ library which provides a set of network simulation models implemented as C++ objects and wrapped through python.

The users using this library must write a C++ or a python application to instantiate the set of simulation models. These objects are needed to:

- set up the network simulation scenario of interest
- enter the simulation mainloop
- exit when the simulation is completed

In Figure 4.1 is shown a scheme of the NS-3's architecture.

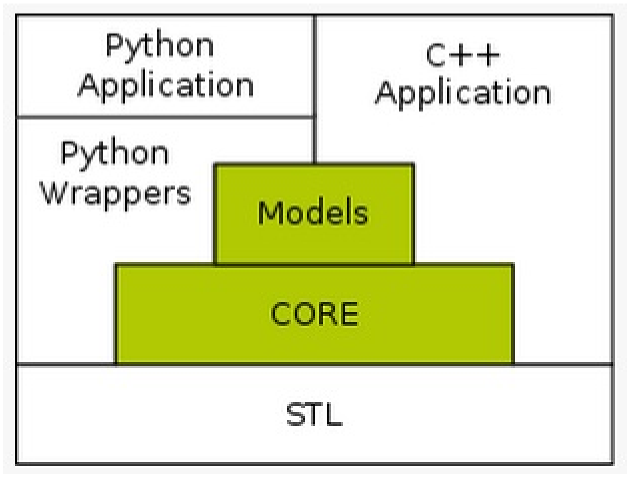


Figure 4.1: NS-3's architecture

4.3 High level design

NS-3 differs from other discrete-event network simulator for the following high level design goals:

- C++ and Python emphasis

  The simulators often use a domain-specific modeling language to describe models and program flow whereas NS-3 uses C++ or Python granting to the user the full support for each language.

- Callback-driven events and connections

  Instead of using specialized "handler" functions that centralize the processing of events in each simulation object, NS-3 associates to an event a callback function. These functions are scheduled to be executed at prescribed simulation times. The use of these callback functions reduces the compile-time dependencies among simulation objects.

- Flexible core with helper layer

  NS-3 provides a powerful low-level API that allows to the users the ability to tune a proper configuration. On top of this layer there is a set of "helper" layer APIs that offers easier-to-use functions with default behaviour. The users can mix both low-level API and "helper" API to develop its own network.

- Emphasis on emulation

  The simulation design is oriented towards use cases that allow the simulator to interact with the real world. ns-3 packet objects are stored internally as packet byte buffers (similar to packets in real operating systems) ready to be serialized and sent on a real network interface. Several different simulation-in-the-loop and virtual machine integration frameworks have been developed, and ns-3 experiments have been carried out on wireless testbeds.

- Emphasis on software reuse

  In NS-3 is possible to create a network to test real application without requiring changes on the application code. This can be done with the use of Linux containers.

- Alignment with real-world interfaces

  The NS-3 nodes are patterned to reflect as much as possible the Linux networking architecture. This better facilitates code reuse and improves realism of the models, and makes the simulator control flow easier to compare with real systems.

- Configuration management

  The NS-3 simulator features an integrated attribute-based system to manage default and per-instance values for simulation parameters. All of the configurable default values for parameters are managed by this system, integrated with command-line argument processing, Doxygen documentation, and an XML-based and optional GTK-based configuration subsystem.

- Lack of an IDE

  The project does not maintain an IDE (Integrated Development Environment) to configure, debug, execute, and visualize simulations in a single application window, such as found in other simulators such as GNS 3 (Graphic Network Simulator 3). Instead, the typical workflow is to work at the command line and integrate configuration and visualization tools as needed.

4.4 NS-3's interaction with the real world

In this section it is explained the way in which NS-3 can interact with the real world.

NS-3 is a simulator that allows the users to simulate the behaviour of a network composed by simulated nodes. These nodes are provided by NS-3 of the following features:

- applications to generate traffic
- net devices and channels to move the traffic

In this way all the components of the system, both computing nodes and network, are simulated but the users can make interact these components even with real entities.

It is important to highlight the fact that these computing nodes are very simple, and in particular are simpler than virtual nodes created with some virtual machine.

In Table 4.1 is represented all the possible configurations available to the user.

Nodes\Networks	**Real**	**Simulated**
Real	your computer and network	NS-3 TAP
Simulated	NS-3 EMU	NS-3 native simulation

Table 4.1: NS-3 configurations

With the term real node we indicate a node not emulated by the NS-3 simulator.

In the following subsections let's look at the different configurations shown in the previous table.

4.4.1 Real nodes and real networks

When both the network and the nodes are real, NS-3 has no use, so this case is not relevant for the analysis of NS-3.

Instead in the other three configurations NS-3 plays an important role.

4.4.2 Simulated nodes and simulated networks

It is the classic configuration of NS-3. In this case the behaviour of both the nodes and the network are simulated by NS-3.

4.4.3 Simulated nodes and real networks

This configuration requires the use of nodes simulated by NS-3 and the use of a real network.

This type of configuration is not very common but still remain useful when used in particular situations.

NS-3 is used with this configuration for the following purposes:

1. to validate a network model
2. to test a simulated application

When we build a network model that tries to predict the behaviour of a real network we must have some reasons to believe that the results thus obtained are correct.

In this case we can compare the results obtained using the network model and the results obtained using a real network.

This process is called model validation and can be accomplished with this configuration.

Another way in which it may be useful to use this configuration is when we want to test a simulated application.

If the simulated applications can correctly run over a real network means that these applications are correct.

In either situations, the use cases are addressed by the NS-3 EMU net device. EMU is from network EMUlation.

Network emulation is a term that refers to the ability to allow the simulator to inject traffic into a real network and vice versa.

4.4.4 Real nodes and simulated networks

The configuration with real nodes and a simulated network can be very useful to test real applications without requiring the deployment of a real network that can be complicated and expensive.

In this type of configuration we want to run real applications on virtual computers that talking over a simulated network.

The grade of virtualization of these computers can be very different. Technically, what we are talking about is platform virtualization.

The platform virtualization can be subdivided in full virtualization and paravirtualization.

4.4.4.1 Full virtualization

In full virtualization a virtual machine environment is designed to completely simulate some underlying hardware and OS (Operating System) environment.

In this case it is possible to create many virtual machines on the same real machine. In particular these virtual machines may have different OS than the real machine OS that hosts them.

For example on a Mac OS system we can create a virtual machine with either Linux or Windows or Mac OS.

This type of virtualization is a heavyweight system because in this case is simulated every aspect of the target machine, maintaining the virtual machines so created isolated from each other.

Hence, the full virtualization has a very high computational cost.

The full virtualization systems available to the user are for instance VMWare and VirtualBox.

4.4.4.2 Paravirtualization

In paravirtualization is created a virtual machine environment without completely simulate the underlying system.

This type of virtualization is more lightweight than the full virtualization and it can be accomplished in two ways:

1. the OS simulated is ported to a virtualization environment
2. the underlying host OS can be used to create the illusion of virtualization of some parts of the underlying host hardware

The second form of paravirtualization is most lightweight than the first so it is the most commonly used.

The fact that it is lightweight allows to run many instance of virtual (simulated) nodes in NS-3.

However if we want high performance from the NS-3 network we must combine raw power with lightweight virtualization. In particular we must use a server-grade multiple core system running Linux directly and over it to use paravirtualization.

Paravirtualization systems of this kind are for example Linux Containers and OpenVZ.

This type of solution allows only to create a virtual Linux guest systems on a Linux host system. This is because these paravirtualization systems exploit some Linux's characteristics.

4.4.4.3 NS-3 TapBridge

Full virtualization and paravirtualization schemes may be combined to suit individual needs. Since ns-3 is a Linux-based system, it is used a Linux mechanism to implement the required functionality.

In particular the mechanism used is the NS-3 TAP mechanism. This uses an ad-hoc net device called TapBridge:

Tap coming from the tun/tap device which is the Linux device driver used to make the connection from ns-3 to the Linux guest operating system, and Bridge since it conceptually extends a Linux (brctl) bridge into ns-3.

4.5 Using real applications in NS-3

As we saw in the Subsection 4.4.4 there are two ways to run a real application in real nodes connected by NS-3 network, full virtualization and paravirtualization.

The choice of which virtualization to use depends on the type of application we want to run and the computational power at our disposal.

Therefore these factors influence the performance required from the simulated network.

In this thesis we are testing an application to discover all the capacities of the links in a path via an Android smartphone and, from the Chapter 2, come to light that it is important not the value of the capacity of the links but the ratio between them.

Furthermore the machine we want to use to run NS-3 do not have much computational power.

From these considerations we choose to use a paravirtualization for the NS-3 network. In particular we use Linux Containers to create the virtual machine environment.

Linux Containers does not create a virtual machine, but rather it provides a virtual environment (container) that has its own process and network space.

The properties of the container are specified in its “*.conf” file.

These properties consist of:

lxc.utsname the username in the container

lxc.network.type the network virtualization used in the container

lxc.network.flags an action to do for the network. This usually is set to “up”

lxc.network.link the interface used for the real network traffic

lxc.network.name the name of the interface seen in the container

lxc.network.hwaddr the interface MAC address

lxc.network.ipv4 the interface IPv4 address

lxc.network.ipv6 the interface IPv6 address

In this case we want that this container is capable of interacting with the NS-3 network, so we must use for the lxc.network.type a network type called veth, and for the lxc.network.link an ethernet bridge previously created.

When the network type used is veth, it is created a new network stack where a peer network device is instantiated and attached, on one side, to the container and, on the other side, to the ethernet bridge specified in lxc.network.link.

In Figure 4.2 is shown a representation of the container and its connection with the host OS.

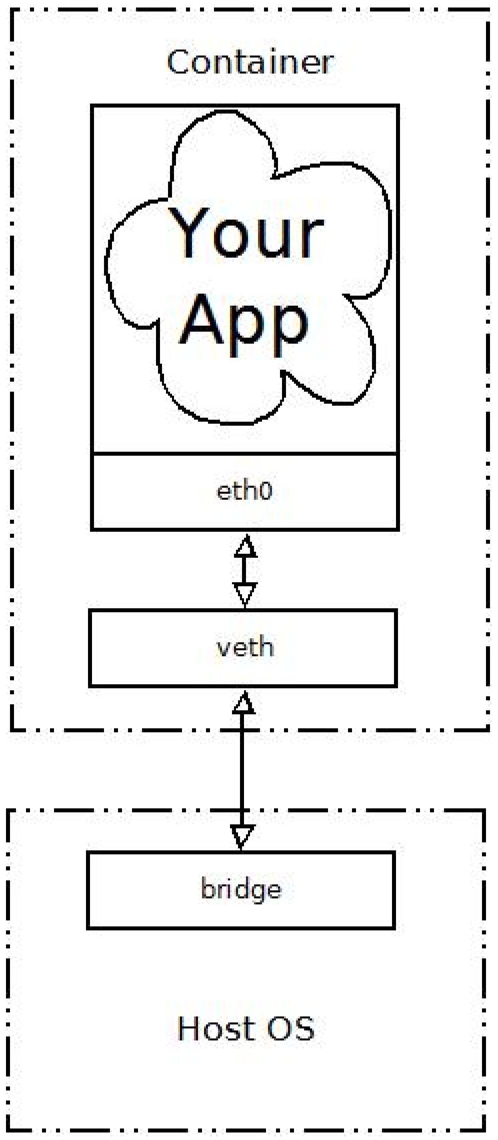


Figure 4.2: Container's connection with the Host OS

In this way we set the container of our application, so now we must create the NS-3 network and make the application communicate with the real world via the simulated network.

In particular we want to run in the container the server application because in this thesis the client lies in an Android smartphone. This means that the NS-3 network must position itself between the container and the real network where is the Android smartphone.

Hence, in the host OS we must create a tap device that NS-3 will use to get packets from the bridge into its process and we must add this tap device to the bridge.

Finally we must create and run a properly configured NS-3 network scenario.

In the NS-3 network, the configuration of both the node logically attached to the container and the node connected with the real network are very interesting. These two nodes have two NetDevice.

In the node attached to the container the first NetDevice is the TapBridge NetDevice, that is a bridge used to make it appear that a real host process is connected to a NS-3 net device.

The second NetDevice for this node depends on the type of network we want to emulate. In our experiments we used only Ethernet networks so the NetDevice used in this case is the CSMA NetDevice.

In the node connected with the real network we must use a NetDevice for the communication toward the NS-3 network and an EmuNetDevice to connect this node, and then the entire NS-3 network, to the outside networks.

The EmuNetDevice is able to send and receive packets over a real network because it make use of an interface configured in promiscuous mode. In particular EmuNetDevice opens a raw socket and binds to that interface.

It is performed MAC spoofing to separate simulation network traffic from other network traffic that may be flowing to and from the host.

In Figure 4.3 is shown a full example of the configuration of the system.

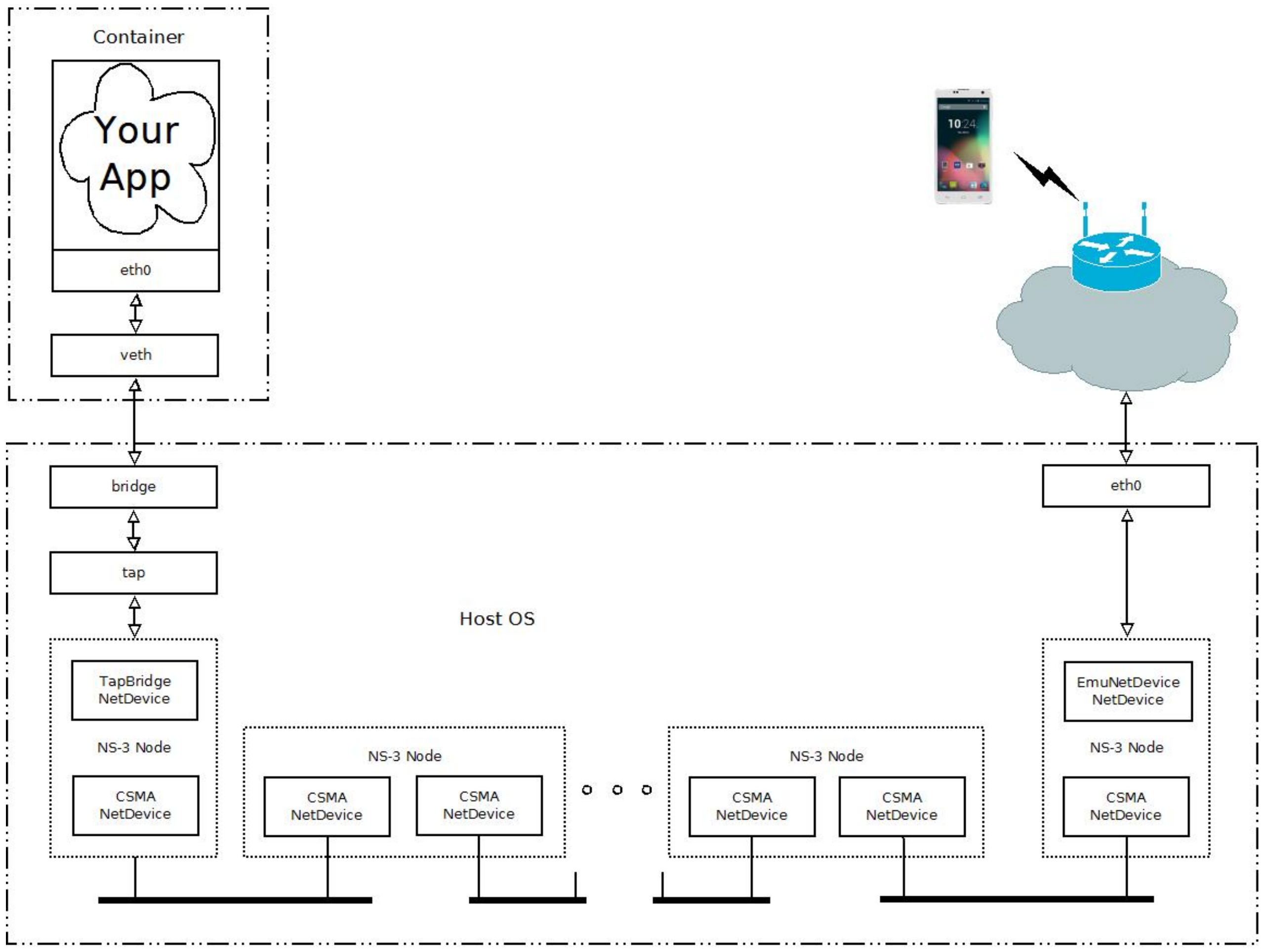


Figure 4.3: System's configuration

Chapter 5

Experiments

In this chapter, we analyze the outcomes of the experiments to evaluate the quality of the results obtained with this algorithm's implementation.

The chapter is structured into three parts. Firstly, we analyze the performance of the simulator NS-3 in different configurations. Secondly, we test our tool using the configuration of NS-3 chosen. Finally, we show the results obtained in a real environment.

5.1 Limits NS-3

In this section we will analyze the performance that can be achieved by the simulator NS-3.

In all our experiments we use on client side a Sony Xperia U running Android 4.0.4 and on server side a generic computer.

We will use NS-3 to simulate a network, so that we can configure the capacity of each link, and a Linux Container to allow the server side application to interact with this network.

As we described in Chapter 4, Linux Containers and NS-3 can work together only in a Linux environment.

This means that if we want to use a computer with a different OS, we must use a virtual machine running Linux.

In Figure 5.1 is shown the scenario used when we use a virtual machine running Linux. Figure 5.2 shows the scenario when we use a computer running Linux.

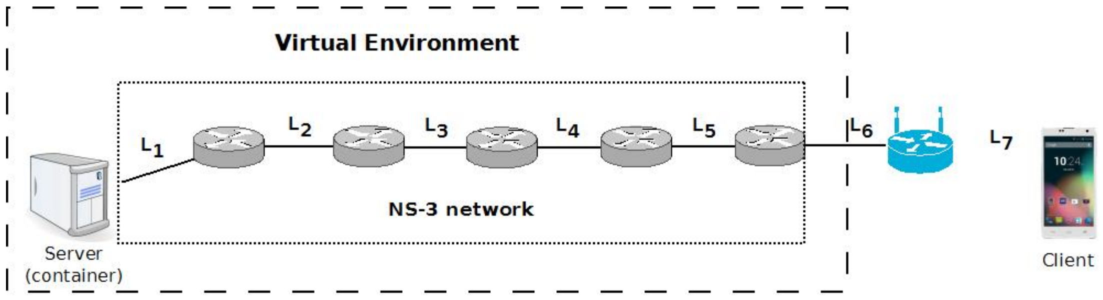


Figure 5.1: Network scenario with virtual machine

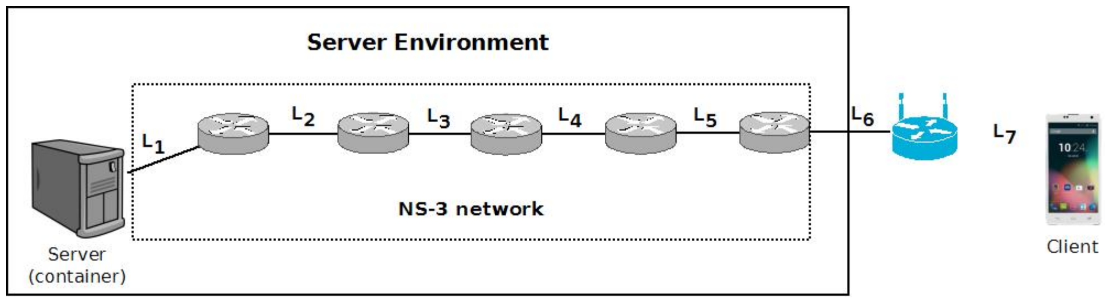


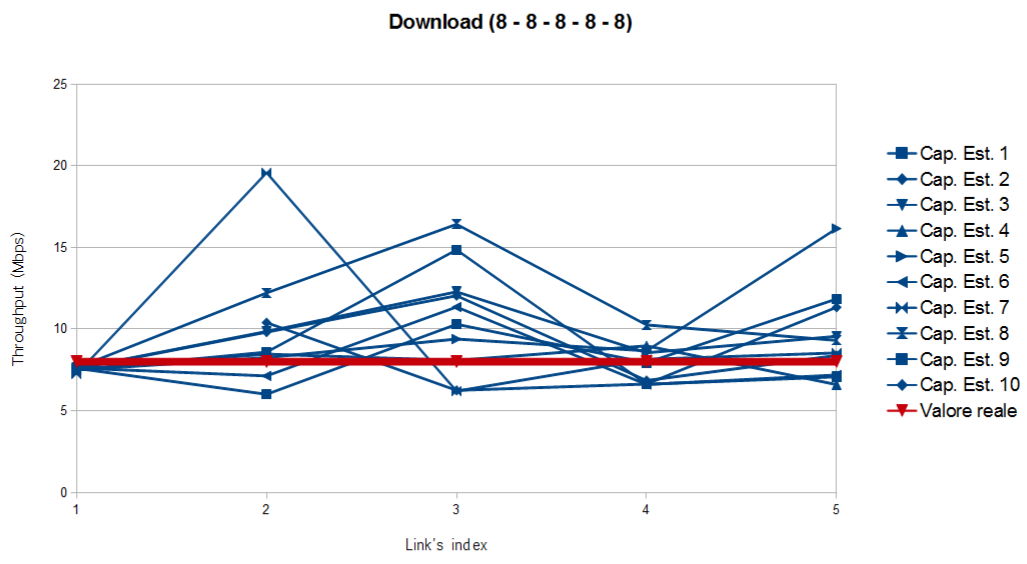
Figure 5.2: Network scenario with a computer running Linux

The performance of the simulator in these two cases is different. When we use a computer running Linux we obtain a better performance than when we use a a virtual machine running Linux.

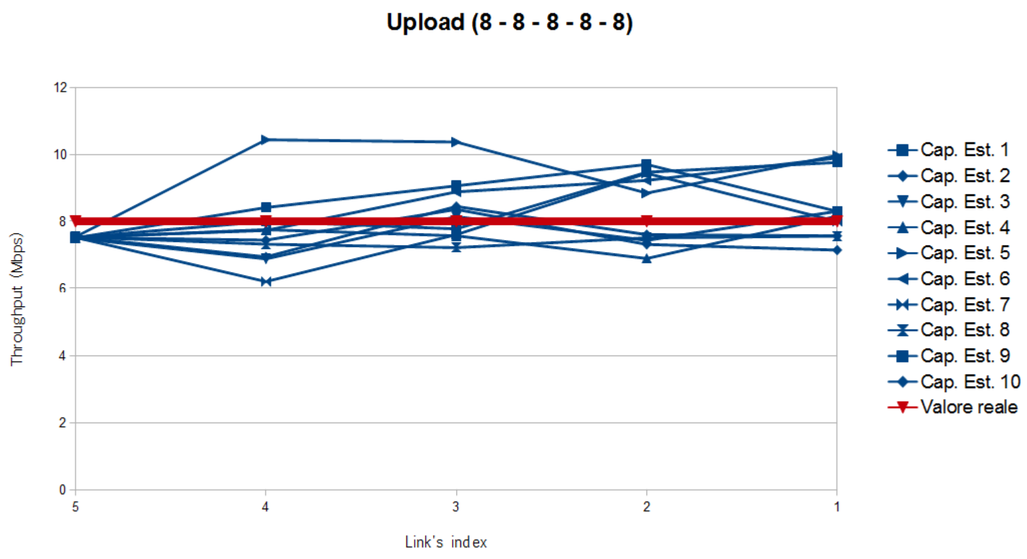
The interesting thing is that the performance will degrade in both cases even when the capacity to emulate are relatively low.

In Figure 5.3 we show the results obtained in the first configuration, both download and upload, when the capacity of each link in the NS-3 network is set to $8\,Mbps$, $r = 1$ and $TS = 45$.

It is important to highlight that these experiments are not affected by cross traffic.



a)



b)

Figure 5.3: Estimations performed using a virtual machine. a) download b) upload

In this case the estimations in both download and upload are not reliable. Indeed, the capacities estimated are barely close to the expected values.

The estimation for the first link is accomplished with the prefix path estimation technique. The estimations for the other links are made with the target subpath estimation technique.

Only the first link is correctly estimated. This happens because the prefix path estimation technique is more robust than the target subpath estimation technique.

The explanation is that the prefix path estimation technique depends on a time interval bigger than the target subpath estimation technique.

The interarrival time seen at the receiver with the prefix path estimation can be expressed in the following formula

$$\Delta_1 = \frac{r\,(s\,(p)+s\,(m))\,(TL-1)}{b_1}$$

Instead the interarrival time seen at the receiver with the target subpath estimation is the following

$$\Delta_i = \frac{r\,(s\,(p)+s\,(m))}{b_{1,i-1}} + \frac{s\,(p)}{b_i} - \frac{s\,(m)}{b_{i-1}}$$

with $2 \leq i \leq 5$, if $TL > 2$ then $\Delta_1 > \Delta_i$

Hence, for the simulator NS-3 is easier to emulate the interarrival time Δ_1 rather than Δ_i

This explains why the capacity of the first link is correctly estimated.

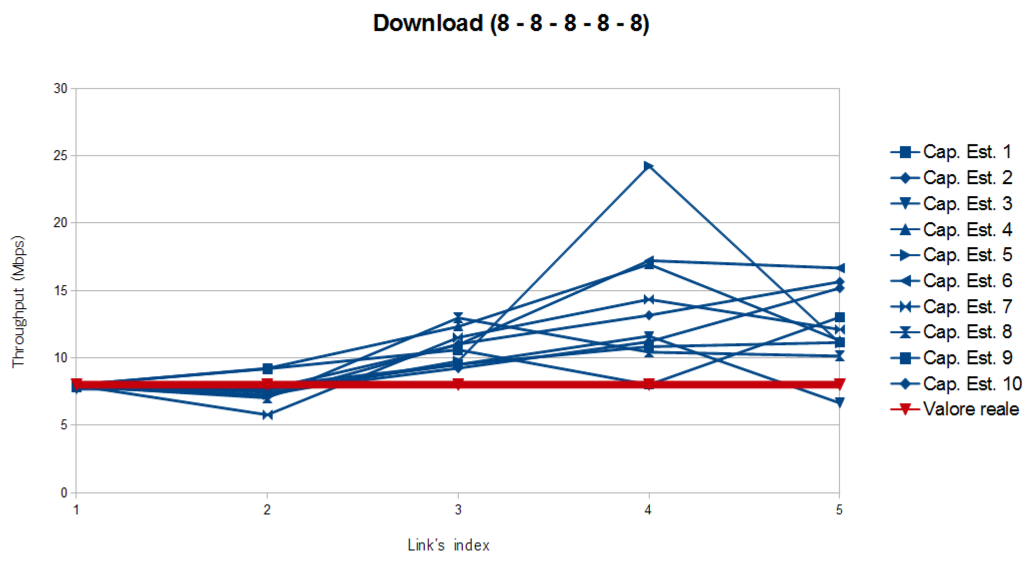
In Figure 5.4 we show the results obtained in the other configuration, both download and upload, when the capacity of each link in the NS-3 network is set to $8\,Mbps$, $r = 1$ and $TS = 45$.

We can note that despite the network has the same configuration, the results are different.

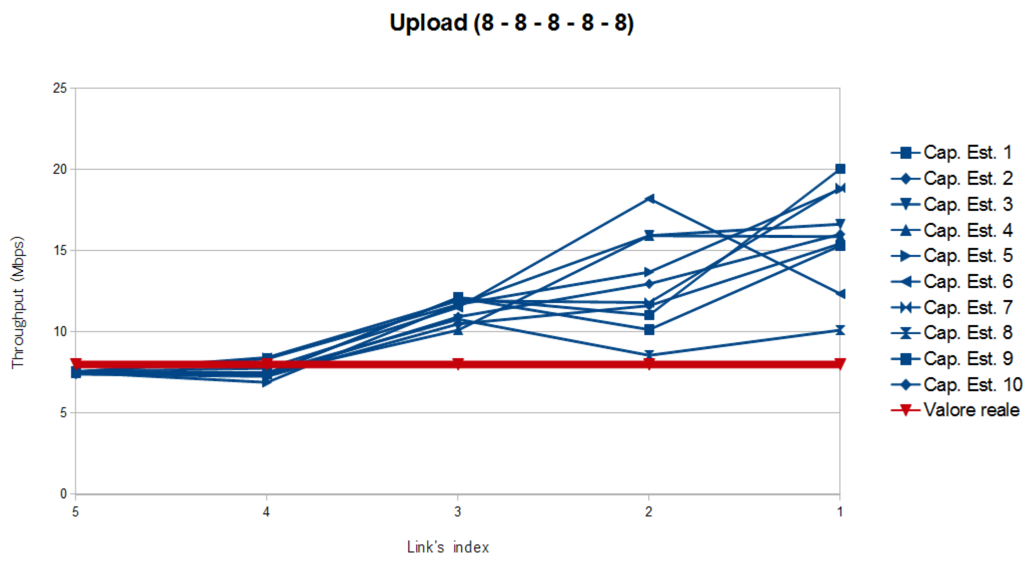
In this case we can see that, in both the estimations, we have a first part where the capacity is correctly estimated and another part where there is a strong overestimation.

This means that even in a configuration without using a virtual machine the simulator NS-3 affects the result of the estimations.

In the following scenarios we use smaller capacity values to be sure that the simulator does not affect the estimations.



a)



b)

Figure 5.4: Estimations performed without using a virtual machine. a) download b) upload

5.2 NS-3 on Linux host

In this section we describe how our tool is exploited, on server side, by a PC and, on client side, by a Sony Xperia U running Android 4.0.4.

The configuration used in these experiments is that described in Figure 5.2.

We used a configuration without cross traffic to evaluate the results of the algorithm in the ideal case.

More realistic experiments are made in Section 5.3.

It is important to note that in this configuration we can only set the capacity values of the links in the NS-3 network and not the links L_6 and L_7.

For these links we can assume that $b_6 = 100\,Mbps$ and $b_7 = 25\,Mbps$ because they are a Ethernet connection and a wireless 802.11g connection respectively.

As a result we will not consider these links because we cannot set their capacity values.

In the following subsections we test our tool in various scenarios setting $r = 1$ and $TS = 45$ if not specified.

The value of the train size can change accordingly to the capacity used in the network.

The value of the train length is chosen on the basis of the results obtained in [2] when the smartphone is connected to a 802.11g link.

These scenarios differ for the values assigned to each link in the NS-3 network.

5.2.1 Increasing values

In this scenario with set the capacity of the links to an increasing values.

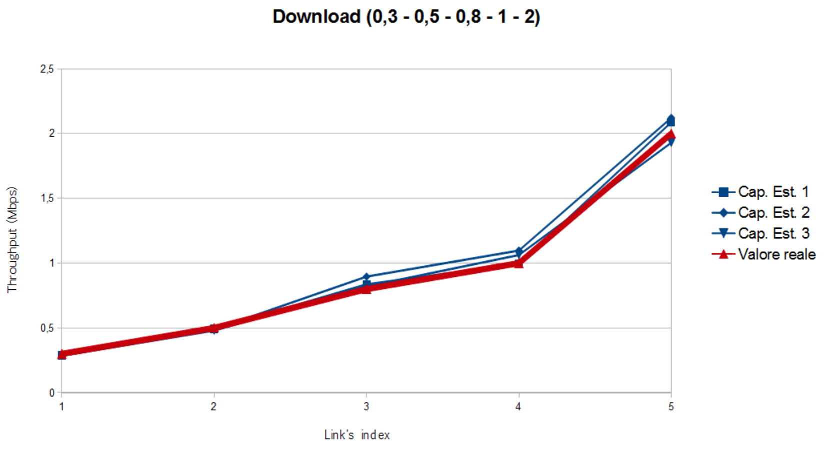
In particular we set $b_1 = 0,3\,Mbps$, $b_2 = 0,5\,Mbps$, $b_3 = 0,8\,Mbps$, $b_4 = 1\,Mbps$ and $b_5 = 2\,Mbps$.

Figure 5.5 shows the results obtained in this scenario.

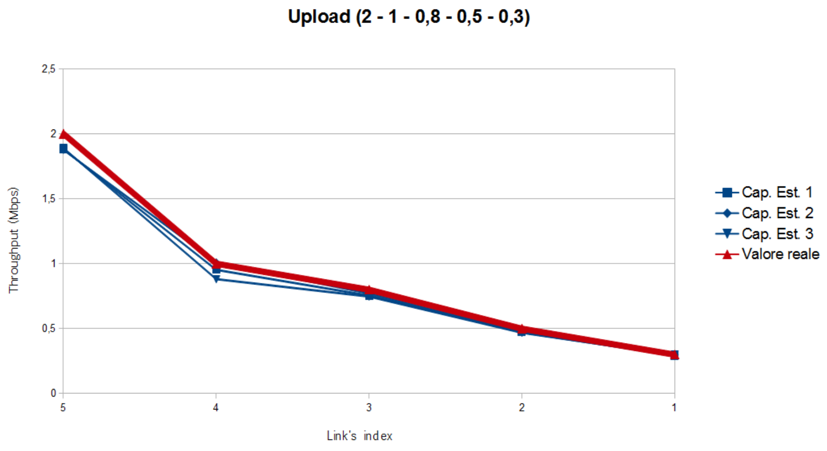
As we can see from this figure the estimations in both download and upload can be considered reliable.

This demonstrates that the tool is able to effectively estimate the capacity of all the links in a path.

At this point it is clear as in the previous case the results were affected by the simulator.



a)



b)

Figure 5.5: a) estimation in download b) estimation in upload

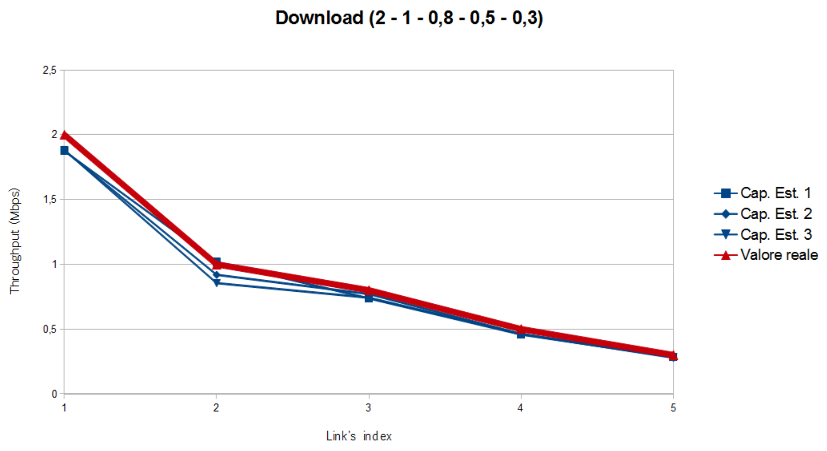
5.2.2 Decreasing values

Conversely to what we saw in the previous subsection we set the capacity of the links to a decreasing values.

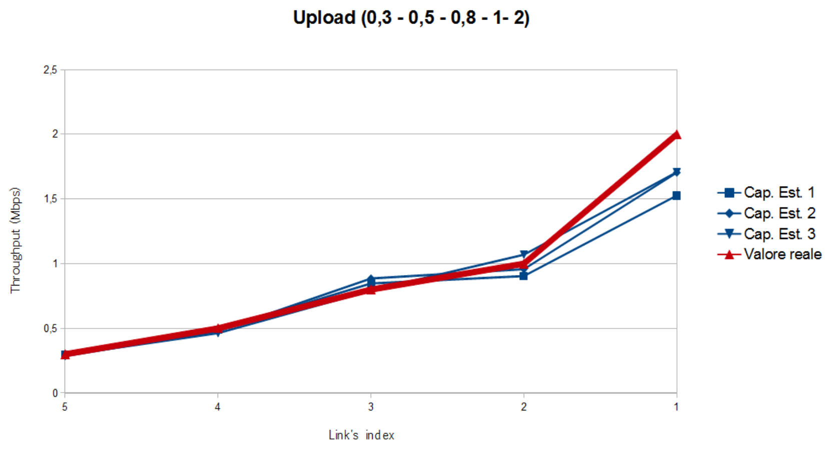
We set $b_1 = 2\,Mbps$, $b_2 = 1\,Mbps$, $b_3 = 0,8\,Mbps$, $b_4 = 0,5\,Mbps$ and $b_5 = 0,3\,Mbps$.

Figure 5.6 shows the results of the estimations performed.

The results are the same obtained in the previous subsection.



a)



b)

Figure 5.6: a) estimations in download b) estimations in upload

5.2.3 'V' configuration

In this case we set the capacity of the links in the first part to decreasing values and in the last part to increasing values.

The capacity values are: $b_1 = 1\,Mbps$, $b_2 = 0,75\,Mbps$, $b_3 = 0,5\,Mbps$, $b_4 = 0,75\,Mbps$ and $b_5 = 1\,Mbps$.

This brings together the characteristics of the scenarios presented in Subsection 5.2.2 and Subsection 5.2.1.

In this way the bottleneck will be in the middle of the NS-3 network.

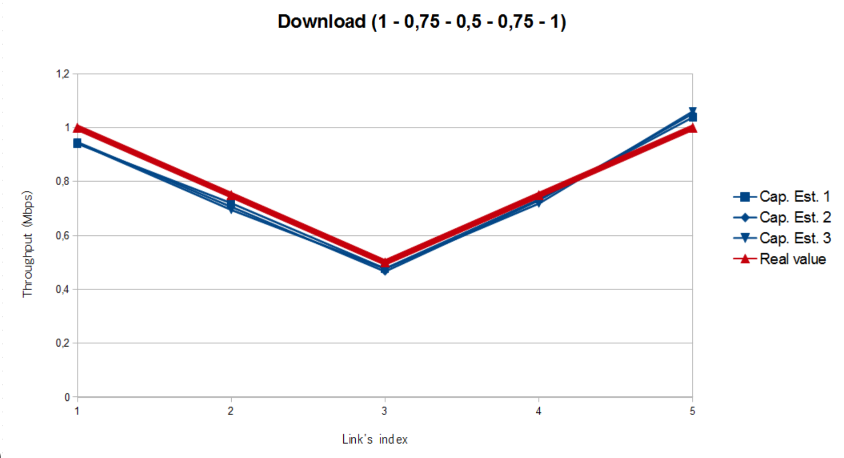
This network configuration will be called for its form 'V' configuration.

Our tool will use the prefix path estimation technique for the first part of the path, and the target subpath estimation technique for the last part to estimate the capacity in this scenario.

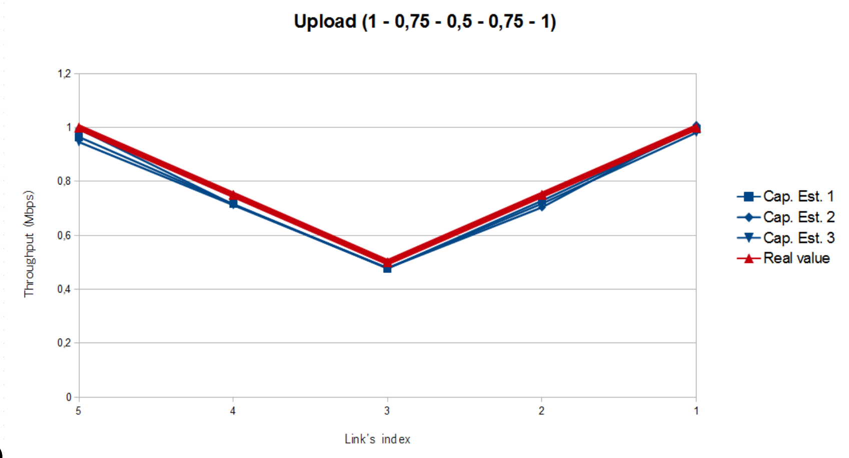
Hence, this scenario allows us to test both the techniques used in this algorithm.

In Figure 5.7 are shown the results of our estimations.

These experiments show that even when the probing traffic must pass through the bottleneck, we can correctly estimate the capacity of the links before this point.



a)



b)

Figure 5.7: a) estimations in download b) estimations in upload

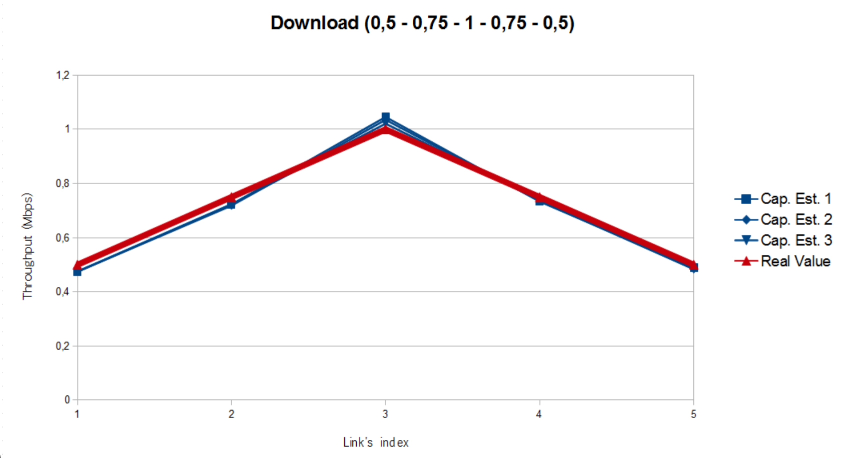
5.2.4 Reversed 'V' configuration

In this scenario the capacity of the first part of the links is set to increasing values and the last part to decreasing values.

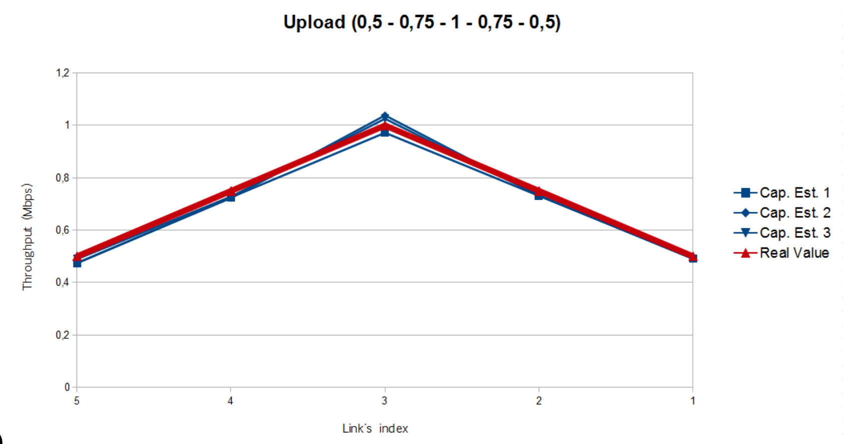
We set $b_1 = 0,5\,Mbps$, $b_2 = 0,75\,Mbps$, $b_3 = 1\,Mbps$, $b_4 = 0,75\,Mbps$ and $b_5 = 0,5\,Mbps$.

The results are shown in Figure 5.8.

It is clear that both in download and upload the estimations can be considered reliable.



a)



b)

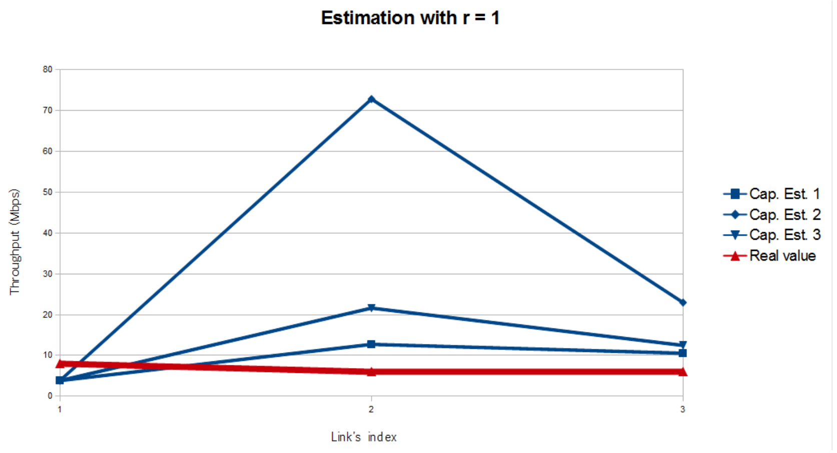
Figure 5.8: a) estimations in download b) estimations in upload

5.2.5 Train size test

The scenario used in this case was configured with $b_1 = 8\,Mbps$, $b_2 = 6\,Mbps$, $b_3 = 6\,Mbps$ and $b_4 = 0,2\,Mbps$.

This scenario is designed to force the use of a train size value greater than 1. Indeed, in this case the condition of the Corollary 2.3 is respected only with $r \geq 2$.

In Figure 5.9 are shown the results obtained using $r = 1$ and $r = 2$.

a)


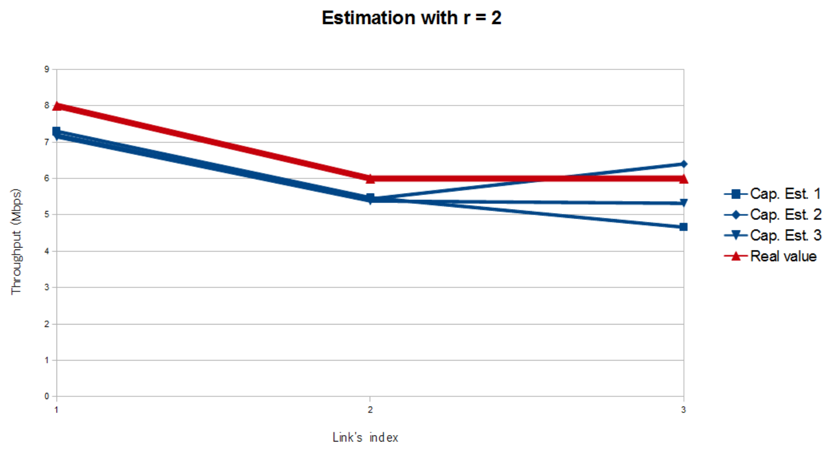
b)


Figure 5.9: a) estimations with $r = 1$ b) estimations with $r = 2$

It is clear that the estimations performed with $r = 2$ are more reliable than the estimations performed with $r = 1$.

This demonstrate the effectiveness of the condition expressed in Corollary 2.3.

It is important to highlight the fact that our tool will independently choose the right value of train size.

5.3 Internet measurement experiments

Finally, we test our tool in a real Internet path. We installed the server application on a server of the National Research Council (CNR) and the client on a Sony Xperia U running Adroid 4.0.4 connected to the wi-fi network of the university of Pisa.

The CNR is a public organization; its goal is to carry out, promote, spread, transfer and improve research activities in the main sectors of knowledge growth and of its applications for the scientific, technological, economic and social development of the country.

Figure 5.10 shows the path used for these experiments and Table 5.1 shows the *a priori*-known capacity values of the links in this Internet path.

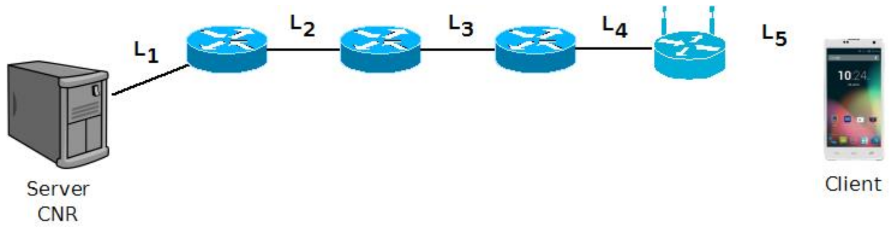


Figure 5.10: The Internet path used in these experiments

b_1	$1\,Gbps$
b_2	$1\,Gbps$
b_3	$100\,Mbps$
b_4	$100\,Mbps$
b_5	$25\,Mbps$

Table 5.1: Capacity values of each link in the Internet path

It is important to highlight the fact that this is an Internet path so it is traversed by the cross traffic. This means that the cross traffic can affect the estimations.

Analyzing the capacity of the links in the Internet path we can see that, in download, to respect the conditions of the algorithm we must use $r \geq 30$.

For this reason we start the estimations in download using $r = 30$.

In Figure 5.11 is shown the results obtained in the estimations in download.

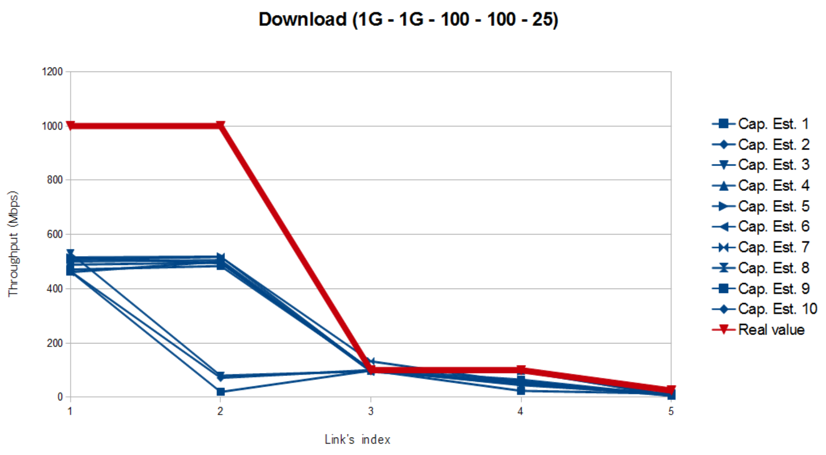


Figure 5.11: Estimations in download of the Internet path

As we can see from the previous figure we do not correctly estimate the first two links. In particular the capacity estimated in these links is about $500\,Mbps$.

By further analysis we found that the maximum value of the sending rate of the sender application is about $500\,Mbps$. This is the reason why we do not correctly estimate these links. Furthermore this overestimates the estimations of the following links when it is used the target subpath estimation.

Instead the other links are correctly estimated the most of the time. Analyzing the cases where the estimations are not correct we found another interesting feauture.

The incorrect estimations are perfomed using the target subpath estimation, instead the correct estimations are performed using the prefix path estimation.

In this cases we underestimate the capacity so the effect of the incorrect estimation of b_1 and b_2 is opposed by the cross traffic.

This demonstrates that the target subpath estimation is not robust in the presence of cross traffic.

Figure 5.12 shows the results obtained in the upload estimations and Figure 5.13 shows the same results but in a zoomed graphic.

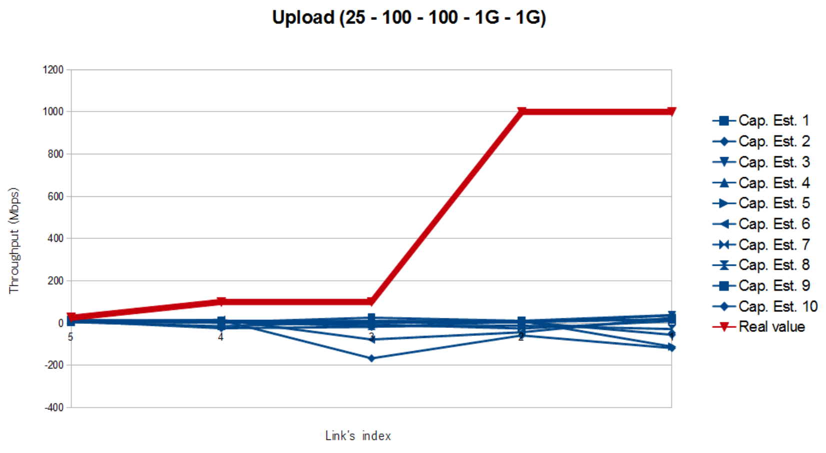


Figure 5.12: Estimations in upload of the Internet path

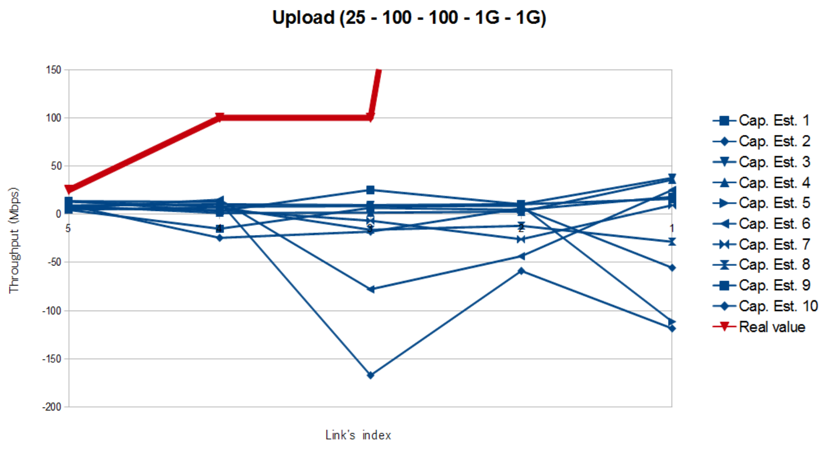


Figure 5.13: Estimations in upload zoomed

The results are very far from the expected values. In many cases the capacity estimated have negative values.

In this kind of configuration the estimations of b_2, b_3, b_4 and b_5 are performed using the target subpath estimation.

We saw in the download estimations that this type of estimation is strongly affected by the cross traffic.

In this case we estimate negative values for the capacity, as we can see in Figure 5.13.

5.4 Conclusions

The experiments showed that this tool in an environment free of cross traffic is able to correctly estimate the capacity of each link in a path.

The estimations are valid as long as the Tailgating Property and the condition in Corollary Corollary 5.2 are satisfied.

In a real environment, however, there is also the cross traffic. It is unthinkable to find a path without it.

In these cases we saw that only the prefix path estimation leads up to a correct estimation. The target subpath estimation is unreliable.

This means that only the estimation in download can be correct because in this case it is very likely that the bottleneck is at the end of the path.

Hence, in this kind of path it will be used the prefix path estimation.

On the other hand in the estimations in upload the bottleneck is placed at the beginning of the path.

Hence, the tool will use the target subpath estimation. This will not correctly esteem the capacity of the links.

Furthermore, the prefix path estimation is affected by the low sending rate of the server application (about $500\,Mbps$).

This means that only the links with a capacity lesser than $500\,Mbps$ can be estimated.

5.5 Future researches

Future improvements for this tool mainly involved the target subpath estimation and the server application.

We must make the target subpath estimation less sensitive to the cross traffic.

Furthermore, the server application must be improved. In particular we must assure that in download the sending rate is at least $1\,Gbps$. In this way we can be reasonably sure that the prefix path estimation will be correct.

In download we saw that in most cases to estimate the capacity of a link is used the prefix path estimation. In upload it is the exact opposite.

In this case it is used the target subpath estimation. This means that unless we improve this kind of estimation, the estimation in upload cannot be reliable.

Acknowledgments

First and foremost, I would like to make a special thanks to my supervisor Prof. Luciano Lenzini for having proposed to work on the development of a tool on a smartphone Android.

Deepest gratitude is also due to my other two supervisors, Ing. Alessio Vecchio and Ing. Enrico Gregori, whose guidance and assistance was very important in this thesis.

Special thanks also to Alessandro Improta and Nilo Redini for giving me access to the server at CNR, and Carlo Vallati for helping me with the simulator NS-3.

Certainly, I also want to thank all of my colleagues of the "*Red Lab*" that have come and gone in recent months: Giovanni, Enrico, Alessandro, Gloria, Gordon, Dario and Nino.

Last but not least I want to thank all my friends at Livorno and my family for supporting me in this years.

Thank you.

Bibliography

[1] K. Harfoush, A. Bestavros, J. Byers, “Measuring Capacity Bandwidth of Targeted Path Segments”, in IEEE/ACM Transactions on Networking (TON), vol 7, pp. 80-92, Feb. 2009.

[2] F. Disperati, D. Grassini, E. Gregori, A. Improta, L. Lenzini, D. Pellegrino, N. Redini, “Smartprobe: a Bottleneck Capacity Estimation Tool for Smartphones”.

[3] J. C. Bolot, “End-to-end packet delay and loss behavior in the Internet”, in Proc. ACM SIGMCOMM’93, Sep. 1993, pp. 289-298.

[4] R. L. Carter and M. E. Crovella, “Measuring bottleneck link speed in packet switched networks”, Performance Evaluation, vol. 27&28, pp. 297-318, 1996.

[5] K. Lai and M. Baker, “Measuring link bandwidths using a deterministic model of packet delay”, in ACM SIGCOMM’00, Stockholm, Aug. 2000.

[6] K. Lai and M. Baker, “Nettimer: A tool for measuring bottleneck link bandwidth”, in Proc. USITS’01, Mar. 2001.

[7] R. Prasad, M. Jain, C. Dovrolis, “Effects of Interrupt Coalescence on Network Measurements”, in Passive and Active Network Measurement (PAM), vol 3015 of Lecture Notes in Computer Science, pp. 247-256, 2004.

[8] B. M. Leiner, V. G. Cerf, D. D. Clark, R. E. Kahn, L. Kleinrock, D. C. Lynch, J. Postel, L. G. Roberts, and S. Wolff, “A brief history of the internet”, SIGCOMM Comput. Commun. Rev., vol. 39, pp 22-31, Oct. 2009.

[9] J. Postel, “Internet Protocol”, RFC 791 (Standard), Sept. 1981. Updated by RFCs 1349,2474.

[10] A. Faggiani, E. Gregori, L. Lenzini, S. Mainardi and A. Vecchio, "On the feasibility of measuring the Internet through smartphone-based crowdsourcing", in WiOpt, pp 318-323, IEEE, 2012.

[11] E. Gregori, L. Lenzini, V. Luconi and A. Vecchio, "Sensing the Internet through crowdsourcing", in Proceedings of the Second IEEE PerCom Workshop on the Impact of Human Mobility in Pervasive Systems and Applications (PerMoby), May 2013, pp. 248-254.

[12] Y.-H. Chu, S. Rao, H. Zhang, "A case for end-system multicast", in ACM SIGMETRICS'00, Santa Clara, CA, Jun. 2000.

[13] J. Jannotti, D. Gifford, K. Johnson, M. F. Kaashoek, J. O'Toole, Jr., "Overcast: Reliable multicasting with an overlay network", in Proc. OSDI 2000, San Diego, CA, Oct. 2000.

[14] D. Andersen, H. Balakrishnan, M. F. Kaashoek, R. Morris, "Resilient overlay networks", in SOSP 2001, Banff, Canada, Oct. 2001.

[15] J. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications", in ACM SIGCOMM'01, San Diego, CA, Aug. 2001.

[16] J. Byers, J. Considine, M. Mitzenmacher, S. Rost, "Informed content delivery across adaptive overlay networks", in ACM SIGCOMM'02, Pittsburgh, PA, Aug. 2002.

[17] J. Kangasharju, J. Roberts, K. W. Ross, "Object replication strategies in content distribution networks", in Proc. WCW'01: Web Caching and Content Distribution Workshop, Boston, MA, Jun. 2001.

[18] P. Radoslav, R. Govindan, D. Estrin, "Topology-informed internet replica placement", in WCW'01: Web Caching and Content Distribution Workshop, Boston, MA, Jun. 2001.

[19] V. Jacobson, "Pathchar: A tool to infer characteristics of Internet paths". [Online]. Available: `ftp://ftp.ee.lbl.gov/pathchar`.

[20] A. Downey, "Using pathchar to estimate Internet link characteristics", in ACM SIGCOMM'99, Boston, MA, Aug. 1999.

[21] V. Jacobson, Traceroute. 1989 [Online]. Available: `ftp://ftp.ee.lbl.gov/traceroute.tar.Z`.