Master's Thesis

Passive queue size estimation from a middlebox in TCP/IP networks

by

Ángel Santiago González Pinar

Department of Electrical and Information Technology
Faculty of Engineering, LTH, Lund University
SE-221 00 Lund, Sweden

Abstract

The new applications and services made available on the internet in the recent years (as P2P file sharing or VoIP applications) have dramatically changed the usage patterns all over the network. The huge amount of users all over the world connected to Internet makes it necessary to develop algorithms and protocols that can provide good QoS to the customers.

The main protocols of Internet are UDP, TCP and IP. Especially TCP is the protocol supposed to provide a reliable connection and congestion control. Several previous studies have shown that one of the main problems nowadays on Internet is the bufferbloating (huge latency experienced by the packets while transmitting due to the presence of large queues in the buffers). This situation produces that a considerable amount of packets might suffer from a big random delay, causing big jitter as well. This issue is even more relevant in delay-sensitive applications, applications that require a high level of interactivity of the user as VoIP or videoconference.

A more accurate study of the Internet behaviour will help us to attack the problem mentioned above. The solution analyzed in this thesis project is taking advantage of one of the options offered by TCP protocol: **TCP timestamps.** By activating this option, the TCP header includes the time a packet was sent. This allows a more accurate study of the Internet traffic behavior. A new algorithm to study the jitter, delay and queues in networks is implemented and tested. This new algorithm using TCP timestamps option is implemented in Python. In order to confirm its validity, it is subjected to several tests.

The algorithm can used to measure the latency and queuing suffered in different connections, by analyzing different data collections from real networks, and to try to identify some patterns that can help to reduce them in the future. The results given by the new algorithm are more accurate and trustable than, for example, when using RTT calculation.

Our algorithm shows that it is possible to estimate time intervals, networks and servers offering bad QoS with high probability. As this thesis provides results based on measuring the current situation, active actions can be taken according to them to avoid those bad QoS situations.

Acknowledgments

This Master's thesis would not exist without the support and guidance of my two supervisors: Anders Waldenborg, Senior Software Engineer at Procera Networks, who assisted me in the technical part during the thesis duration, by giving me very helpful advices and solutions in the several problems I faced, and Kaan Bür, Researcher and associate teacher at Lunds University, who guided me in the elaboration of the final report, presentation and also helped me in the technical part as well.

I really want to thank the company Procera Networks for giving me the opportunity to develop my Master's thesis in collaboration with them. The experience of being involved in a real project together with a company has been really helpful for me and I am sure will help me in my future work life.

Of course I want to thank to my family, for their continue support, even if we were separated, but they always transmitted me their support.

Last but not least I would like to thank all the people who showed interested on my thesis, or helped me in any moment within these last 5 months.

Lund, May 2013

Ángel Santiago González Pinar

Table of Contents

List of Figures

CHAPTER 1

Introduction

On its journey from source to destination, a packet passes several network components that may queue it in a buffer. The necessity of buffers in Internet can be explained easily. When more packets are sent to the link than this can deal with, there are packet losses and collisions. To avoid it, the user can send packets only when the link is able to transmit without crashing. This causes the packets to "wait" to be sent on the user's side, so there is a need for queuing the packets. Another common situation when buffering happens is when connecting two different links, each of them with a different data rate. If a packet is sent through a high-speed link and arrives at a new link whose speed is lower, it is forced to wait in a buffer until the new link is able to transmit it. The packet might be buffered too when several links are multiplexed to a single one; it must wait until the single link is able to handle it.

Some attempts to avoid packet losses have done nothing but deteriorate even more. For example all over the network there have been set huge buffers to avoid packet loss, but adding bandwidth can actually hurt in terms of latency rather than help. The larger the buffer is, the worse problems are in terms of latency and jitter. This problem is known as *bufferbloating*, and it seems like it is present all over the network [1]. It can be concluded that some packet loss is essential in order to avoid bufferbloat. [2]. The lack of consensus about congestion algorithms (or lack of algorithms at all in some networks) also contributes to magnify the bufferbloating problem [3] [4]. All these problems are even worse in asymmetric links, and this is just a very common link due to the prevalence of ADSL. This mismatch between TCP and asymmetric networks is deeply analyzed in [5] [6] [7] [8].

This bufferbloating problem is quite new, and even though some studies have been done in order to test it and improve it, more accurate methods are required [2]. By inspecting timestamp values in packets sent through a so-called **middlebox** and correlating those, it is possible to calculate relative

changes in propagation delay between the sender and the middlebox. These changes can be interpreted as changes in the amount of buffering and latency.

The previous work within the field had mainly focused on optimizing downlink throughput, setting aside the optimization of **latency**. A deep study of the latency is demanded because it can be the bottleneck of the connection. If there is a network link with low bandwidth, just putting several links in parallel will increase the bandwidth. However, if there is a network link with bad latency, no topology can turn it into a link with good latency. Even links with enough bandwidth can show a bad latency: for example, the fastest link that exists nowadays; if it lacks of good congestion algorithms, or if it is connected to a link with much less BW, experiences a high latency and consequently delivers a poor QoS.

There is always a lower bound of the time that a packet needs to travel from source to destination that can never be beaten, does not matter how small it is. If a large amount of data is sent, this latency is not very noticeable, but if smaller amount of data will be transmitted, then the latency is even higher than the transmission time. Even when transferring big files the computers have to send to each other lots of little control messages, so the performance of small data packets directly affects the performance of everything else on the network [9]. As internet grew, the problem of latency has become more important. Recent studies and experiments have shown that TCP/IP networks suffer from big latency, delays and jitter that sometimes damage the quality of the TCP connection. That's why taking action to solve them is required.

The solution developed through this thesis report is based in an algorithm which takes advantage of the TCP timestamps option in order to calculate delays, jitters and queues in a more accurate way. When both the receiving and sending time of a packet is available at one side, the calculation of these figures becomes more accurate and reliable. The algorithm is coded in a Python program. It works on previous offline collected data files, but it should be implemented in a way that can be easily adapted to live networks analysis. In order to confirm the validity of the code, some tests are performed using different tools and simulators to test different scenarios with a wide range of possibilities. The code also provides some statistics based on the traces from real connections, about how much data and time is required in order to deliver a good result.

This Python code also runs over a representative file with a selection of real connections. This file represents a real scenario that can be found anywhere in TCP/IP networks. It is collected with the help of the middlebox taking advantage of Deep Packet Inspection (DPI) to collect data traces which are interesting for this overall analysis; it contains the widest variety of connections, delays and jitter in order to estimate a proper pattern of QoS in Internet. By defining events or situations that can produce big jitter in Internet connections, it will be possible to avoid latencies in the future, as different actions could be taken depending on which host, location or network is accessed.

The project has been done in cooperation with **Procera Networks**, an American-Swedish company focused on designing and selling middleboxes using DPI technology to measure the QoS of the customer. We have worked on data files delivered by Procera Networks as they have been chosen to show a scenario as representative as possible of a random Internet scenario. This thesis work has also been a contribution to the Celtic-Plus project **IPNQSIS (IP Network Monitoring for Quality of Service Intelligent Support).** This is a European research project whose main objective is to develop monitoring systems in order to study and improve the behavior of Quality of Experience (QoE) by the analysis of networks and their impact in the final customers. [10]

CHAPTER 2

Background and relevant work

Transmission control Protocol (TCP) is one of the core protocols of Internet, and is a protocol in the transport layer (according to the TCP layered model which is based on, although a little different from, the OSI model). TCP handles the establishment of a connection, the sequencing and acknowledgment of packets (in order to order and retransmit the packet when lost or corrupted) and the data flow (controlled by the window size).

TCP uses a *sequence number* to identify each byte of data so that the data can be reconstructed in order. For every byte transmitted, the sequence number must be incremented. A checksum field is also included. In order to guarantee that all the bytes are received correctly and in the same order, TCP implements positive acknowledgment (ACK) with retransmission (the receiver answers with an ACK message as it receives the data, while the sender keeps a record of each packet it sends). The sender also keeps a timer in case an ACK gets lost or corrupted.

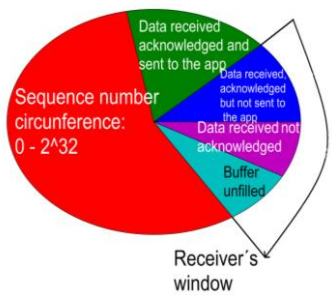

Figure 1: Receive window in TCP

TCP uses a *cumulative acknowledgment* scheme (the receiver sends an ACK meaning that it has received all data preceding the acknowledged sequence number by asking the sender for the sequence number of the next byte it expects to receive).

TCP uses a sliding window flow control method. The *receive window* field represents the amount of additional bytes that the receiver is able to buffer. The sending host can send only up to that amount of data, and when a receiver sends a window size of 0, the sender stops sending data. [11]

2.1 TCP: deficiencies and first modifications

The current scenario where communications are nowadays makes necessary to modify some parts of TCP protocol (mobile Internet, fiber optic speeds, delay-sensitive applications or hundreds of millions of users worldwide for example), as it cannot guarantee the best QoS. That is clearly seen in the fact that modifications have been made to the protocol that have changed the way it reacts to losses and congestion: TCP Reno, TCP Tahoe, or algorithms as Random Early Detection (RED) or Explicit Congestion Notification (ECN) are just some examples that show the necessity to improve the protocol.

The different options that TCP offers are subsequent modifications that show the necessity of changing the protocol in some way. These are: the TCP option to deal with window´s sizes bigger than 16 bits (window scaling option), the SACK option, which uses selective ACK instead of TCP normal cumulative ACK (as the recovery from losses is not either ideal in TCP) and timestamps option, which can be used for several improvements as it is explained throughout Chapter 3. [12]

In the TCP tail drop algorithm, a router or other component buffers as many packets as it can, and drops the ones it cannot buffer. If buffers are constantly full, the network is then congested. This implementation does not provide good QoS, and that´s why some other modifications and algorithms were created.

2.1.1 TCP Tahoe and TCP Reno

To increase the congestion window, TCP uses the "slow start" mechanism. It starts with a window of two times the maximum segment size (MSS). For every packet acknowledged, the congestion window increases by 1 MSS. Although the initial rate is low, it means an exponential increase. When the congestion window surpasses a threshold, the algorithm enters the "congestion avoidance" state: as long as non-duplicate ACKs are received, the congestion window is increased by one MSS every round trip time. The implementation of Tahoe and Reno differ in how they react to packet loss:

- Tahoe: Triple duplicate ACKS are considered the same as a timeout. Tahoe sets the threshold to half the current congestion window, reduces congestion window to 1 MSS, and resets to slow-start state.
- Reno: If three duplicate ACKs are received it reduces the congestion window to half its value, sets the slow start threshold equal to the congestion window, and enters a phase called "fast recovery". In this state, the missing packet that was signaled by three duplicate ACKs is retransmitted, and it waits for an acknowledgment of the entire transmit window before returning to congestion avoidance. If there is no acknowledgment, TCP Reno experiences a timeout and enters the slow-start state.

In case of a timeout, both algorithms reduce congestion window to 1 MSS. [13]

2.1.2 TCP New Reno

TCP New Reno (defined in 2004) improves retransmission during the fast recovery phase of TCP Reno: for every duplicate ACK, a new unsent packet from the end of the congestion window is sent, to keep the transmit window full and maintain a high throughput during the filling of the holes process. For every ACK that makes partial progress, the sender assumes that the ACK points to a new hole, and the next packet beyond the ACKed sequence number is sent. This allows New Reno to fill large holes, or multiple holes, in the sequence space. [14]

2.1.3 Compound TCP

Compound TCP is a Microsoft implementation of TCP (2006). It is mainly designed for connections with large bandwidth-delay products. It uses the estimation of queuing delay as a measure of congestion. Compound TCP maintains two congestion windows: the first window is increased the same way as a TCP Reno one, and the other is a delay-based window. This second window increases rapidly if the delay is small. Once queueing is experienced, the delay window gradually decreases to compensate the TCP Reno one. The size of the actual sliding window used is the sum of these two windows. [15]

2.1.4 RED (Random Early Detection)

RED calculates the average queue size and drops packets based on probabilities. When the buffer is almost empty, all incoming packets are queued. As the queue grows, dropping probability will grow as well. When the buffer is full, no more packets fit (the probability of discarding is 1) and all incoming packets are dropped.

RED is more fair than tail drop as it does not have preference against bursty traffic. The more a host transmits, the more probably its packets will be dropped. [16]

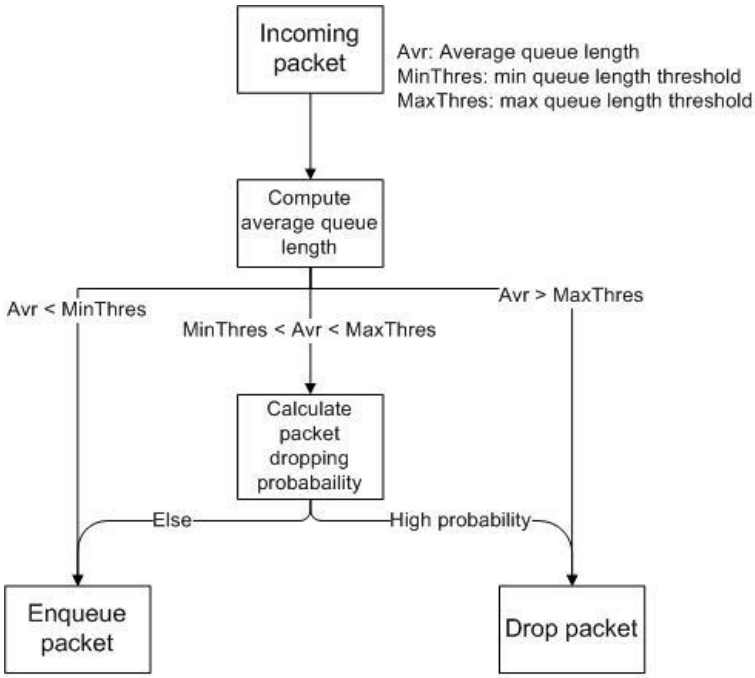

Figure 2: RED algorithm

2.1.5 ECN (Explicit Congestion Notification)

ECN allows end-to-end notification of network congestion without dropping packets. ECN has to be previously configured and supported by both sender and receiver. Once enabled, an ECN-aware router puts a mark in the IP header instead of dropping a packet when congestion is about to happen. The receiver of the packet echoes the congestion indication to the sender, which reduces its transmission rate. [17]

2.1.6 Karn's algorithm

RTT can be difficult to calculate, especially when there is high packet loss or long periods without transmitting. Karn's algorithm was designed to calculate accurate estimations of the RTT when using TCP. RTT is estimated as the difference between the time a segment was sent and the time that its ACK returns. When packets are re-transmitted the ACK could correspond to the first transmission or to any other.

Karn's Algorithm ignores retransmitted segments when updating the round trip time calculation. Round trip time estimation is based only on unambiguous ACK. [18]

2.2 Later developments

Several studies have been carried out to investigate the effect of queuing and latency in the network. In [19] a methodology to monitor the upstream queuing delay experienced by remote hosts is proposed. A deep study on bufferbloating was developed by [20]. It´s based on a passive measuring in a FttH network connecting around 90 homes with a campus (1 GBps). Packet-level traces are recorded following a certain pattern to make it as accurate as possible. The main observations were:

- Buffering is happening to some extent.
- RTT to residential peers is generally longer than to non-residential peers.
- 99.6% of residential RTT samples and 98.3% of non-residential RTT samples are less than one second.

The conclusion was that the queues that impose big delays and bufferbloating (huge buffering of packets inside the network that causes high latency and jitter, as explained in chapter 1) were not found. However, the studies also point at the possibility that a deeper study can indeed find the problem.

2.2.1 Deep Packet Inspection (DPI)

DPI is a promising new tool for analyzing the network. It is a way of packet filtering that analyzes the packets that pass through a component called "middlebox", searching for viruses, spam or any defined criteria, in order to decide whether the packet should pass and with which priority. DPI-enabled devices have the ability to look at "Link" and "Network" layers of the TCP/IP model. DPI can also be configured to look through Layer 2-7. In TCP/IP scenarios, network equipment only needs to analyze the IP header, but for shallow inspection, TCP and UDP headers are also checked. [21]. DPI can help avoiding undesired delays in those applications more sensitive to latency by setting different priorities to the packets. That's one of the main focuses of the company "Procera Networks". [22]

In this thesis only packets chosen thanks to DPI are analyzed in order to keep the confidentiality of the data (it is done by selecting only HTTPS traffic). Doing this way the timestamps values and the rest of the parameters are accessible, but not the data content. However DPI offers a wide range of new possibilities to improve latency. The study by J.Dangaard [8] precisely takes advantage of DPI; is focused on achieving low latency during saturation of the uplink. It also demonstrates how it is possible to achieve full downlink and uplink utilization in asymmetric links, while at the same time supporting different delay-sensitive applications by placing a middlebox between the PC and the network. That prioritizes traffic in function of their "delay-sensitive" level.

Latency-sensitive and some other types of applications require allocating and sharing the link resources between network applications according to their service requirements. To achieve this, it is necessary to control the different traffic that goes into the link, as so the timing and drop probability of the packets can be fairly and efficiently implemented. Modifying the TCP/IP stack is not an option, because is impossible to have full access to all the devices connected to Internet, so the middlebox solution is more convenient and simple technique. The results shown in [8] fulfill the objective of optimizing ADSL connection in terms of interactive comfort and maximum link utilization. Especially the latency suffered by high priority packets has been decreased by the installation of a packet scheduler and by prioritizing ACK packets and the traffic more sensible to latency.

Recent advances have shown that DPI is not only a way to improve latency in TCP/IP networks, but can also provide profits. Spanish mobile operator Yoigo is planning to charge the customers in function of the contents they get, instead of giving flat-rate, or paying for the generated traffic: rather than paying for a volume of data, customers will pay for a video or a song. If the user is listening to streaming music, the price would include the usage of the network and the content itself. [23] [24]. It can be concluded then that DPI can both improve the latency in the network, while offering the companies an interesting new way of pricing.

CHAPTER 3

TCP timestamps

As it has been shown in Chapter 2, TCP does not provide good enough QoS for modern scenarios. The changes in the application profile imply buffer congestion and big latencies in some networks. In order to improve that, this thesis studies the contribution that the usage of **TCP Timestamps option** can have in the avoidance of latency. It is framed in a DPI scenario. TCP timestamps provide us with the statistics needed for a better understanding of the situation, which is the essential first step for latency avoidance, thanks to the access to the time the packet was sent. This passive study allows a better understanding of the latency and implementing different active solutions according to the statistics and results delivered.

The algorithm is based on calculations at the receiver; the delay and jitter suffered by the packets, which is interpreted as a factor to quantify the QoS of current TCP/IP networks in terms of latency. Both the delay and jitter are calculated taking into consideration the time a certain packet arrives at the destination and the time it was sent (extracted from the **TCP timestamp** option field). Finally, the maximum queuing time suffered in the connection is also calculated.

This way of calculating delay, jitter and queues is more accurate than using RTT measurement, which calculates the delay since a packet is sent and its ACK arrives to the sender, so two packets are involved in this transmission and it is less accurate than the algorithm which has been implemented in this thesis.

3.1 Calculation of timestamps

As stated in section 2.1 and the introduction of chapter 3, TCP **timestamps** can help to solve some of the current problems of TCP protocol. **Timestamps** add extra bytes, but reduce unnecessary retransmissions and avoid wrapped sequence numbers.

The timestamp field has the following structure:

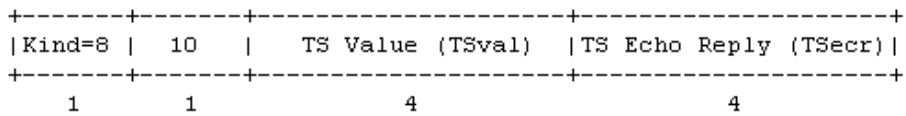

Figure 3: Timestamp field

The 2 main fields are Timestamp Value (TSval) and Timestamp Echo Reply (TSecr). TSval contains the current timestamp value, while TSecr is only valid if the ACK bit is set in the TCP header and it echos a timestamp value that was sent by the remote TCP. TSecr value will generally be from the most recent Timestamp option that was received [12]. The timestamps' clock should not be too slow (it must tick at least once for each 2^{31} bytes sent due to the receiver's algorithm) nor too fast (to avoid a fast wrap in the field vaue). Based on these considerations, the timestamp clock frequency is set in the range 1 ms to 1 sec per tick. In order to compare the timestamps of the destination with the timestamps of the source, a known frequency is needed. This known frequency is a multiplying coefficient "F" set into a range of normalized values. Then, this clock value should be converted into a 32-bits format so the different devices can deal with it, so a round-off is also needed.

The TsVal is calculated by Equation 1:

$$TsVal(n) = floor(F \times Ts(n) + O) \quad \textbf{(Equation 1)}$$

Where Ts(n) is the time (ms) when packet "n" was sent, O is an offset and F is a constant value used during all the connection. This F value is usually normalized to 125, 250, 1000, 1024.

3.2 Use of timestamps to calculate RTT

RTT is difficult to measure, as it can be easily miscalculated due to retransmissions or packet losses. One solution to avoid this is using **timestamps** option. The next image shows one example of this advantage:

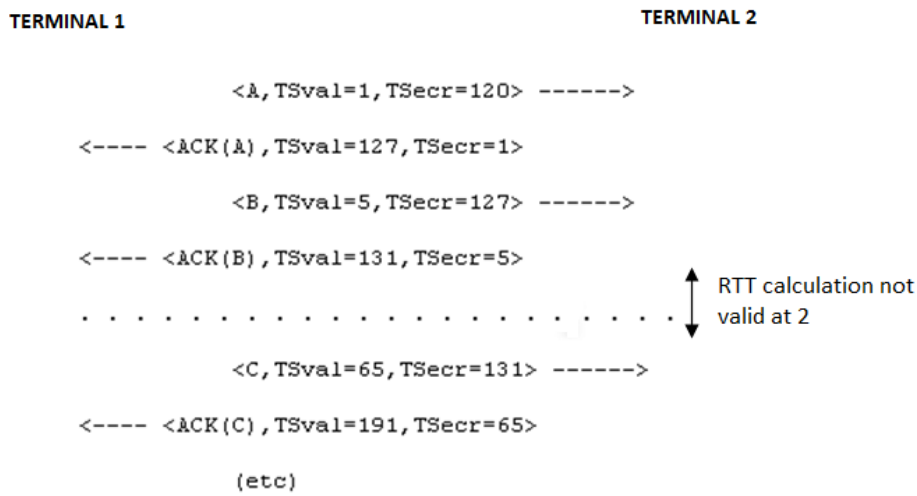

Figure 4: RTT measurement with Timestamps

TSecr value updates averaged RTT only if it acknowledges new data. If there is a big pause without sending anything, RTT calculation won't be done when receiving segment C at Terminal 2 [12].

3.3 Use of timestamps for Protection Against Wrapped Sequence Numbers (PAWS)

It is a mechanism to reject old duplicate segments: for example segments from old TCP connections or with a huge delay that are part of the previous sequence range. A segment is discarded if its **Timestamp** is lower than any other recently received. [12]

CHAPTER 4

Jitter and queue size estimation algorithm

An algorithm to extract TCP timestamps data from trace files and calculate the delay based on that data have been developed. It has been implemented in Python. The following flow chart shows the steps followed.

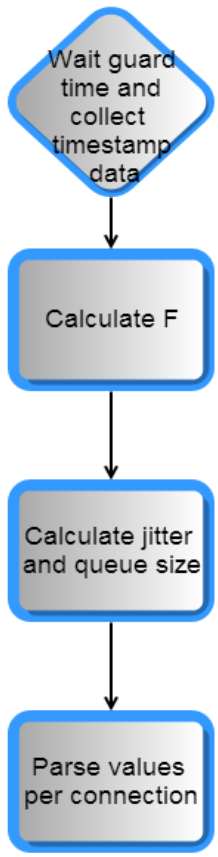

Figure 5: Flow chart of the Python program

The calculation of **F** factor is essential in order to get the rest of the different figures. The necessity of waiting a guard time is explained in the following sections 4.1.1. and 4.1.3. Then the jitter and the queue size are also calculated. Finally, these two parameters are parsed per connection in order to analyze them, as it is explained in chapter 6.

4.1.1 Calculation of jitter

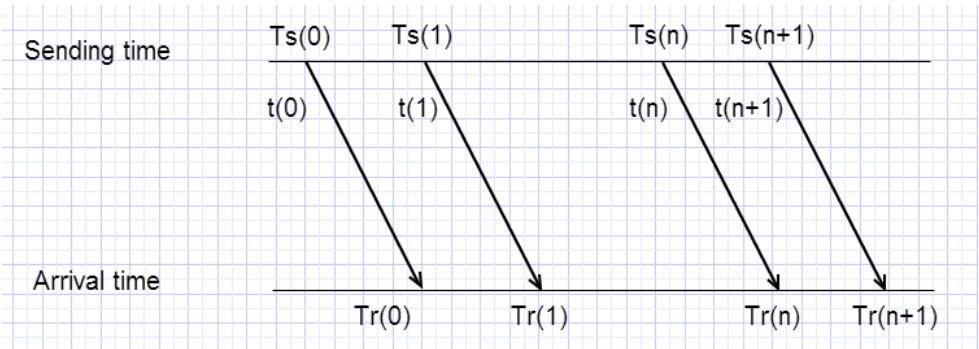

Figure 6: Packet exchange between 2 devices

Figure 6 represents the packet exchange between two devices. Ts(n) represents the times when the packet "n" was sent from the 1st device, t(n) is the TCP timestamp for the packet "n" and Tr(n) represents the time when the packet "n" arrives to the 2nd device.

In order to calculate the delay and the jitter, we need to look at the TCP header. The most important feature is that these calculations use the information provided by TCP Timestamp options. The timestamp field is a function of the sending time, modified by a constant and an offset, and its value it is rounded off to the lower integer value.

$$t(n) = floor(F \times Ts(n) + O) \qquad \textbf{(Equation 2)}$$

There is a necessity to calculate the constant F in order to decode the time where the packet was sent, as it is the sender who sets it and it is unknown for the receiver. The algorithm implements the following equations:

$$Td(n) = Tr(n) - Ts(n) \quad \textbf{(Equation 3)}$$

$$Tr(n) = Ts(n) + \delta_n + j_n \quad \textbf{(Equation 4)}$$

where δ_n and j_n are the transmission and propagation delay suffered by the packet. The delay suffered by the nth packet (Td(n)) is the time gap between the time where the packet has been received and the time it was sent. However the time that the packet was sent is not directly accessible for the receiver, while the time the packet arrives is well recorded. To access the sending time, TCP offers the TCP timestamp option. However, the timestamp is a function itself of the sending time, dependent of a constant and an offset. The offset, as well as the clock synchronization problem, can be eliminated by working with the difference between TCP timestamps from different packets.

$$t(n) - t(0) \cong \big(Ts(n) - Ts(0)\big) \times F + O - O = \\ = (Ts(n) - Ts(0)) \times F \quad \textbf{(Equation 5)}$$

This calculation is more accurate as the packets are more separate in time, as the error in the calculation due to the lower round off of the timestamp becomes really small. The next approximation is also assumed, which will be explained with an example in the next chapter:

$$Tr(n) - Tr(0) = Ts(n) + \delta_n + j_n - Ts(0) - \delta_0 - j_0 \cong \\ \cong Ts(n) - Ts(0) \quad \textbf{(Equation 6)}$$

because the δ_n and j_n values are small when compared to the difference between the packets' arrival times. The constant F can then be estimated:

$$F \cong \frac{(t(n) - t(0))}{(Tr(n) - Tr(0))} \quad \textbf{(Equation 7)}$$

The jitter is the difference in delay between two consecutive packets.

$$Jitter(2 \to 1) = Td(2) - Td(1) = \\ = \big(Tr(2) - Ts(2)\big) - \big(Tr(1) - Ts(1)\big) = \\ = \big(Tr(2) - Tr(1)\big) - \frac{1}{F}(t(2) - t(1)) \quad \textbf{(Equation 8)}$$

The value of both approximations in Equation 6 and Equation 7 is more reliable as the packet "n" and the packet "1" chosen to calculate F factor are more separated, because the errors due to the **disregarding of the δ** delay and the **lower round-off in the F** calculation become less important.

4.1.2 Calculation of queue size

The algorithm to calculate the maximum queuing that any packet has suffered in one connection is shown in the picture below:

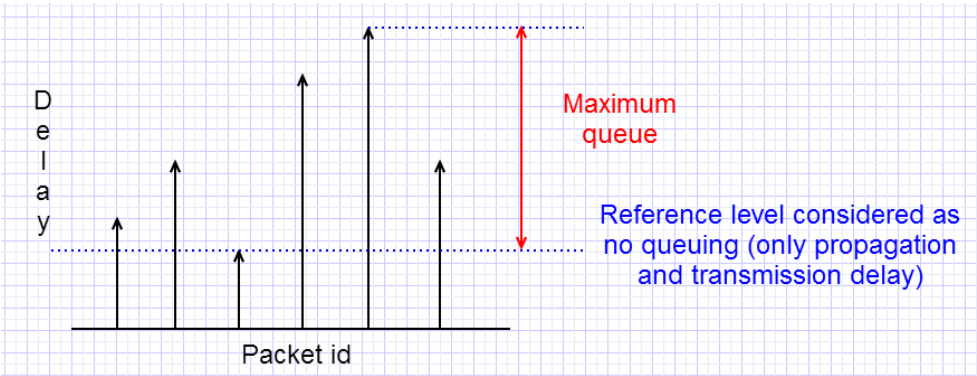

Figure 7: Graphical representation of the maximum queue time

Each arrow represents the delay suffered by a packet transmitted in a connection. The packet that experiences the smallest delay, is considered as reference level (we assume that this packet did not suffered from any queuing; it is just affected by the propagation and transmission delays). The difference between the packet that experiences the maximum delay and the one that presents the smallest delay is considered as the maximum queue in the connection.

$$Maximum\ queue = \max_{\forall\, i,j \in connection} (delay(j) - delay(i)) \qquad \textbf{(Equation 9)}$$

4.1.3 Calculation of guard time

As it was said in section 4.1.1, the calculation of F is more accurate as the two packets are more separated in time, because the errors due to round-off and disregarding of small delays become insignificant.

This is why we first run our algorithm to calculate the most accurate estimation of F, taking the first and the last packets in the .pcap file. However, it is not possible to adopt this approach in a live network environment, as in a live connection the last packet is never available to the programmer, because packets are arriving at random intervals and not ordered. Thus, a second version of the same algorithm has been developed, which can operate in real-time. The main idea is to calculate the F for every new single packet that arrives with respect to the first one, and keep updating this value: this improves the calculation of F as packets that arrive later in time are more separated to the first one. However, the first packets that arrive drive into a miscalculation, where the F value can be far away from the nominal one. That's why there should be a lapse of time until the algorithm starts to calculate it.

This interval without calculating the F and therefore without registering the jitter and queue (guard interval) is determined by looking at several connections and making a statistic to set a time gap from which the F factor will be very probably very close to its nominal value, in any connection that is studied. The guard interval can be expressed either in time or in number of packets, depending on the situation that want to be applied.

In order to gather these statistics that determine the guard time to be waited by the algorithm, a deep study of a collection of many different connections was done. This collection was a file that is deeply explained in section 4.2 (many-connections.pcap). Then the F factor was calculated using the timestamp differences between the first packet and every other one by one. It must be noted that the F value is set individually by each end node creating TCP packets, so it needs to be calculated per connection. The F nominal value is the F factor when calculating it between the first and the last packet (as the first version of the code, since the calculation using the most separated two packets gives the most accurate result). Figure 8 shows both the F vs. time and the error in F calculation vs. time for one sample connection.

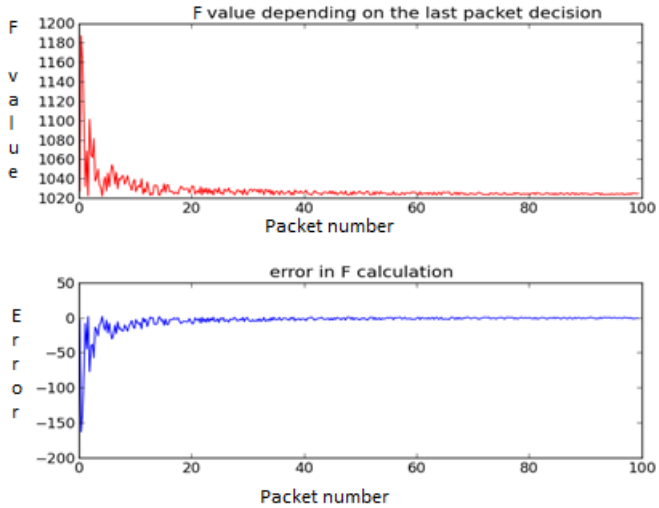

Figure 8: F vs packet distance for one connection

Running this code over several connections, the packet margin is calculated as the first packet that delivers an error within 5%:

$$-0{,}05 < error < 0{,}05$$

The margin in time is obtained as the difference between the arrival time of this packet and the arrival time of the first packet. Once both margins are calculated over a considerably big number of connections (the connections present in the file many-connections.pcap, which is explained in section 4.2), the general guard interval is calculated by looking at the statistical distribution of these two margins.

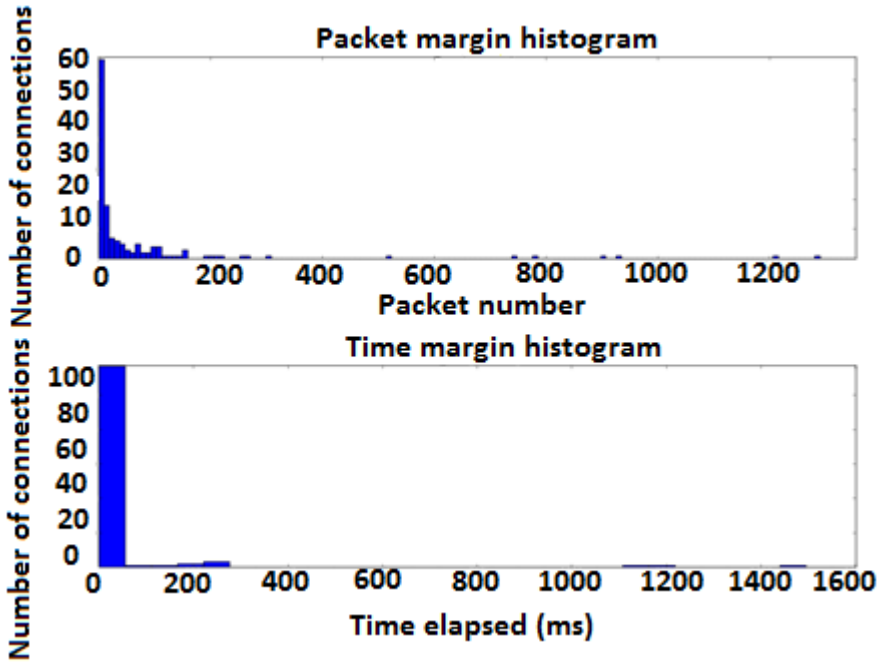

Figure 9: Packet margin (up) and time_margin (below) in F calculation (zoomed)

The mean value of both margins is not accurate to obtain the guard interval, because a big margin in one of the connection while the others have lower margins result in a big average margin that does not reflect the reality. So the Cumulative Distribution Function (CDF), and the histogram of the margins calculated for each connection, lead to a better approximation. The CDF is determinant in order to establish the margin as accurate as possible. Figure 10 shows the CDF function of the packet and the time margins:

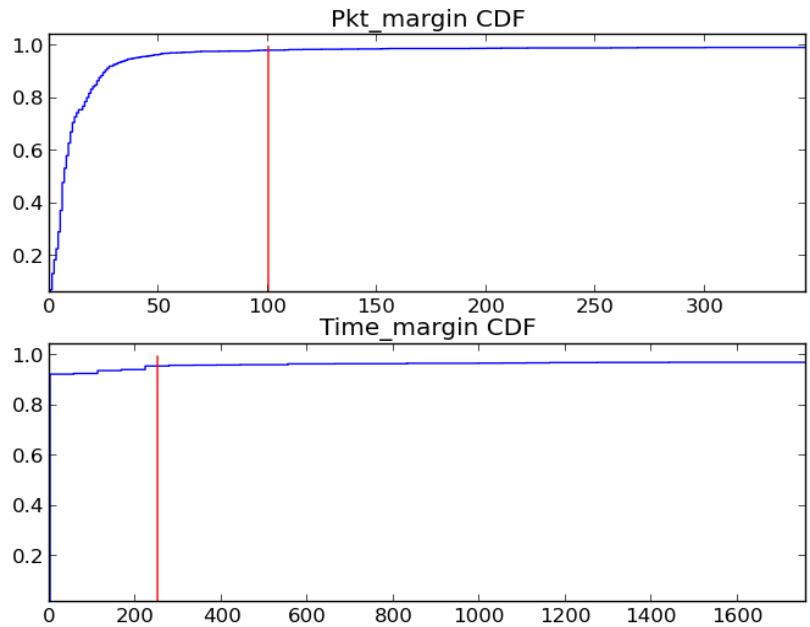

Figure 10: CDF of packet and time margins (zoomed)

Looking at the plot, the **packet margin** was decided to be **100,** while the **gap time** is set to **250 ms**. Both red lines show the percentage of connections that have an insignificant error in F calculation when:

- Waiting 100 packets: ~ 98 %
- Waiting 250 ms ~ 96%

It is also important to emphasize that some of the connections that have huge packet or time margins, never deliver a proper F value, and cannot be treated as part of the study. As working with real data, some corrupted connections appear (for example, wrap in the TCP timestamp field can happen). The packet margin determines that only connections with more than 100 packets should be considered. The calculation of F factor is then delayed 250 ms after the first packet arrived.

Once the F factor has been calculated, the next step is to get the rest of the figures, following the process described at Figure 5.

4.2 Files containing connection data

We want to cover a range of connections that can represent an Internet environment as accurately as possible. However there are some limitations due to confidentiality and privacy issues. Even if the focus of the project is to improve QoS in Internet, is not possible to access the full data exchanged in the different connections. For this reason, all the trace files contain only HTTPS traffic (TCP port 443), as it encrypts the traffic. Looking at HTTPS .pcap files, we can have full access to all the information required for the algorithm (to the headers mainly) but not to the data itself. As only traffic from the port 443 is being analyzed, some other connections are being discarded. This analysis is more accurate in operative systems that enable by default TCP timestamps and use HTTPS (i.e. Android) and less accurate when these options are disabled (i.e. iOS).

The data that were used to start testing the code that analyzes the latency in TCP/IP networks is from capture files delivered by Procera Networks. These data contain multiple connections corresponding to the traffic going through a middlebox at their office (equipped with the DPI technology they develop that selected interesting traffic: for example, data exchange when accessing both a wired and a wireless network). Several .pcap files were taken because the study of the latency required many connections to provide a good overview. The main file used to calculate the guard time needed for the algorithm is called **many-connections.pcap**. This file contains data from 1,809 connections, exchanging 110,682 packets. In this file the data were selected in a way that every packet has TCP timestamps option activated. This connection records packets for a period of 57,740 seconds (16 hours)

In order to analyze interesting real data measurements, the file **capture-20130404-123423.pcap** is used. This file contains traces of connections to both *wired and wireless networks*, as well as several connections to the same host. This will be explained more in detail in Chapter 6. In this .pcap file 6,138 connections are stored, which involve the transmission of 384,334 packets, from where 78.61 % use TCP timestamps option. The capture file lasts for 5,392 seconds (1 hour and a half). An important fact is that the algorithm only runs the algorithm over the packets that have TCP timestamp option activated. The rest are not taken into consideration.

Finally, the biggest collection of connections data is stored in 39 files zipped in **parsed_capture-2013*.pcap.gz**. These pcap files are the ones used to represent a normal Internet scenario. All the 39 files cover a total amount of 109,712 connections, 18,421,426 packets, and the percentage of packets using TCP timestamps is 34.3 %. The time duration of all the files together is 3,630 s (one hour more or less). The most interesting feature of these data files is that it contains traces of several connections to the same host; the study of the correlation of these various connections will show very useful results.

CHAPTER 5

Validation of the algorithm

Once the algorithm is programmed in Python, it is necessary to test it before analyzing real traffic and gathering statistics of the current Internet. The algorithm itself is a new way of calculating delays, so it must be proved that it is valid.

As explained in Chapter 4, the algorithm runs over previously captured data files; the way of testing is then generating capture files according to different configurations. The main parameters (Tr(n), Ts(n) and t(n)) are set to well-known values, so the jitter is expected to be in certain range; once the file is generated, the algorithm shows the results and comparing them to the expectations according to the configuration, we can determine if the code is working as expected. In order to isolate the tests from the way of generating the capture file, three different ways of doing so are used, which are explained in the next sub-sections. In this part of the Thesis, the use of Linux OS was mandatory, as it is especially interesting for connection issues. When the code has successfully passed all the tests, its robustness can be confirmed and real traffic can be analyzed.

5.1 Netem + HTTP Server

Linux OS offers a simple way to connect two Linux machines by the Netem tool, which is run from the command line. First of all, in order to connect both machines, an Ethernet crossover cable is needed. Once they are connected physically, the network should be configured as following:

```
Ifconfig eth0 <ip_adress>
```

This command should be executed in both machines, with different IP addresses (10.1.1.1 and 10.1.1.2 for example). After this step, both computers are able to start communicating with each other. This transmission is ideal (no delay, no packet loss…) but these parameters can be modified using Netem as follows:

Add a parameter:

```
tc qdisc add dev <interface> root netem <parameter><value><variation><distribution>
```

Change a parameter previously defined:

```
tc qdisc change dev <interf> root netem<parameter><value><variation><distribution>
```

Show the parameters used:

```
tc qdisc show
```

Delete a parameter previously defined:

```
tc qdisc del dev <interface> root
```

- The <interface> field should include the one that was configured with ifconfig command (eth0 in this case)
- <parameter> should refer one of the following:
    - Delay
    - Loss (%)
    - Corrupt (%)
    - Duplicate (%)
    - Reorder (%)
- <value> is the desired parameter value
- <variation> adds a ± random value within this field to the desired parameter value.
- <distribution> defines the distribution of the added random value.

The network is already configured as showed in Figure 11, so in order to test it, packets should be sent from one to the other.

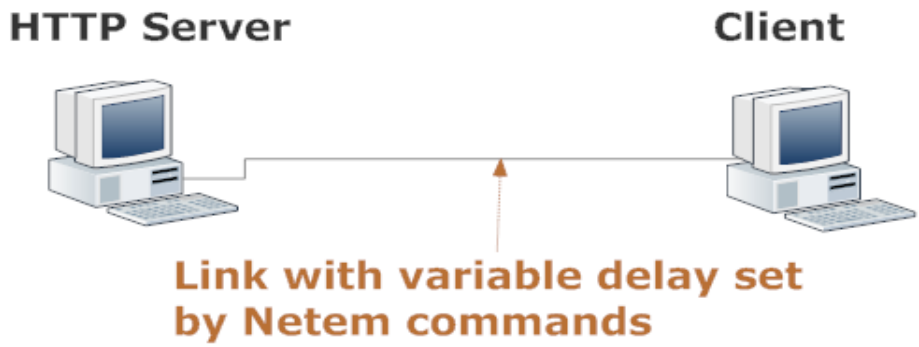

Figure 11: HTTP server + Netem network configuration

First of all one of the computers is configured as a HTTP server as follows:

```
cd /tmp
dd if=/dev/zero of=1M bs=1M count=1
python –m SimpleHTTPServer 8000
```

This creates the server, and the other computer accesses it by typing:

```
wget http://10.1.1.1:8000/1M
```

10.1.1.1 is the IP address assigned with ifconfig command. Once the setup is done and is running, Wireshark collects all the traffic generated in this connection creating the capture file.

With the parameters, different setups are generated to be studied:

- No delay: the delay is set to 0. This means that every single packet only experiences propagation delay, and it is exactly the same for all of them, so the jitter in this simulation should be equal to 0. This is the ideal transmission environment.
- Constant delay: using the Netem commands, the delay is set to a constant value in the range {100, 200, 400, 700}. The jitter should still be 0, as every packet is delayed the same, but the connection time length should become longer as the delay is higher.
- Random delay: in this case, the delay is a random value within certain limits. The jitter shall be present, as the delay fluctuates in time, but it should never be > |2*<variation>| field set in Netem command. The <variation> is set in the range {10, 30, 50, 100} ms.

In order to test if the size of the connection is relevant, every simulation is tested when accessing two different file´s length: 1M and 10M, just changing the bs parameter. The results obtained from the three different configurations are shown and explained below:

5.1.1 No delay

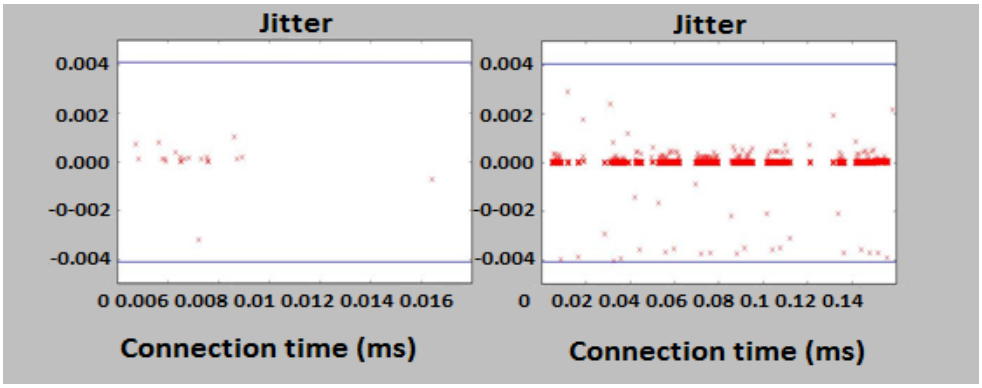

Figure 12: No delay when downloading 1MB to the left and 10 MB to the right

As expected from the theoretical study, the jitter in this case is not present. Most of the spots show a null value (which means that the delay suffered from consecutive packets was exactly the same, in this case only propagation delay). However some spots are more spread, even if the delay was set to a fixed number. This small variation has nothing to do with the delay or jitter, but to the error made by rounding down F value due to (Equation 7) and dismissing of the transmission and propagation delays in (Equation 6). The two blue lines determine 1/F and – 1/F, which are the error margins due to the round-off remains. As explained in Chapter 3, F is a known frequency needed to compare both the sender's and receiver's timestamps; consequently the inverse of this frequency is the smallest time increment we can detect, the time resolution. All the jitter spots are framed in the middle, so the results are as expected.

It can also be detected that the connection time without delay is quite small, and it increases when downloading 10 MB instead of 1 MB.

5.1.2 Constant delay

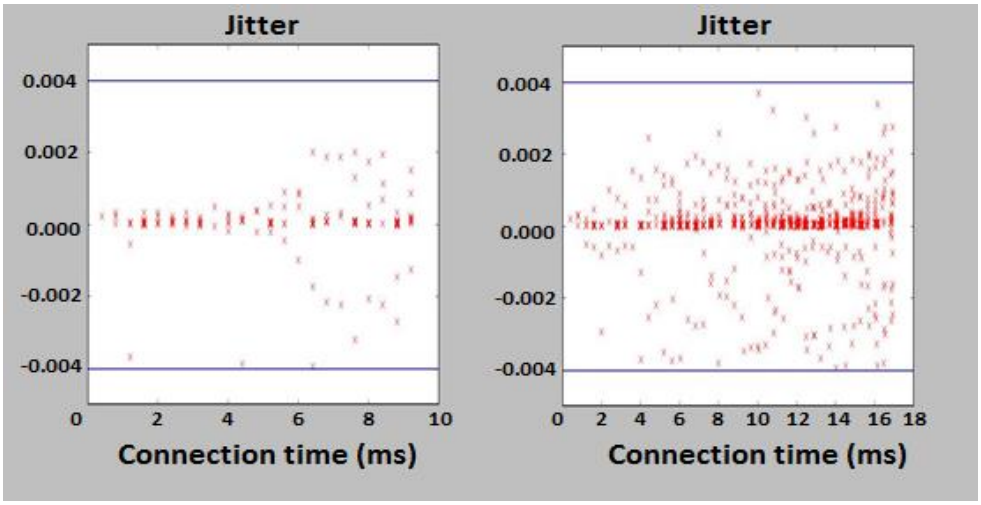

Figure 13: Constant delay when downloading 1MB to the left and 10 MB to the right: the blue lines show 1/F and -1/F.

This case is very similar to the previous one. The fact that constant delay is set to the channel means that every packet suffers the same latency. According to (Equation 8), the jitter should not be present. As in 5.1.1, the deviation of some spots comparing to 0 never exceeds the limits set by 1/F and -1/F, so jitter is not distinguishable from the error we make with the floor function, as expected.

The connection length has increased a lot comparing to the case with no delay. This is a good example to see how in the transmission of a big packet, even a small delay damages the QoS in terms of latency. As stated in [9], this is because a huge amount of small control packets are exchanged even if they are not noticed by the user. This connection length also increases in the download of 10 MB with respect to 1 MB. Figure 13 shows the case where the delays is set to be 700 ms; when simulating with the other values stated in section 5.1, only the connection length changes, but the behavior remains the same.

5.1.3 Random delay

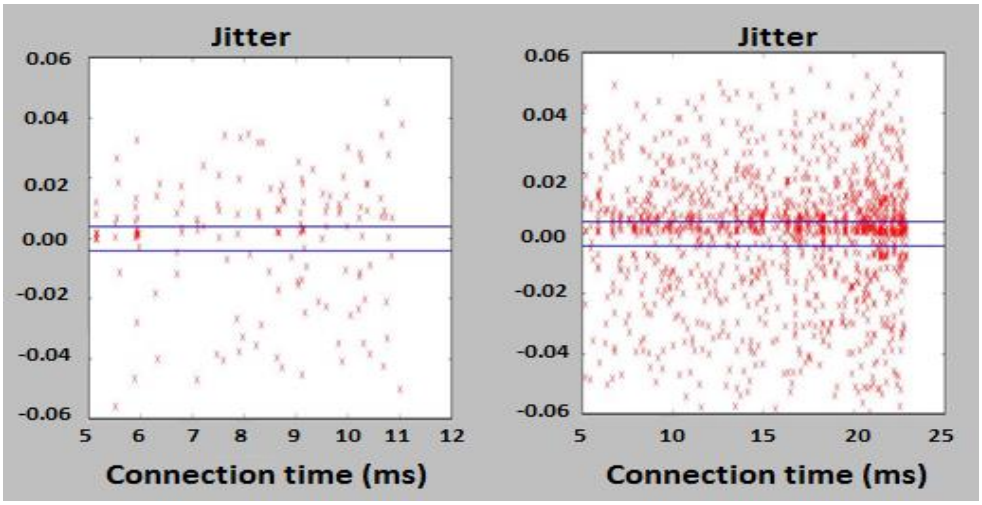

Figure 14: Random delay when downloading 1 MB to the left and 10 MB to the right

This configuration of Netem sets the delay as a randomly added value varying within a range of ±30 ms. In this case every packet suffers from a different delay, so the jitter must be present (spots must exceed the blue lines). However these spots never go over ±60 ms, as this is the maximum jitter that a packet can suffer in the extreme case that the previous one experiences +30 ms and itself -30 ms. The connection length is again longer when accessing a 10 MB, as expected.

A brief summary of the jitter standard deviation values are recollected in the table below:

Case	Jitter st. deviation
Random (variation = 10ms)	7.2 ms
Random (variation = 30ms)	21.4 ms
Random (variation = 50ms)	35.2 ms
Random (variation = 100ms)	78.7 ms

Table 1: Summary of results with Netem

These results show the success of the test, as they match the expectations.

5.2 Python code to generate connections

Another choice to generate a controlled test connection was developing a Python code that creates manually a .pcap file, like filling the gaps of a usual Wireshark .pcap catch. In order to create these .pcap files, the different parameters of the connection are set: first of all, setting TCP, IP and Ethernet. Then assigning the source and destination IP addresses. TCP protocol requires several parameters in order to generate properly its header: these are source port, destination port, window size, sequence number, ACK number or checksum enable as well as some data that is not relevant in this study. The most important parameter that is set in TCP is the TCP timestamp value. This is calculated as explained in 4.2.2:

$$t(n) = floor(F \times Ts(n) + O)$$

O value is chosen as 123456, F = 1024, and Ts(n) is set in different ways depending on the focus of our test connection. The time the packet is sent (Ts(n)) always increases 0.25 seconds for every single packet with respect to the previous one (packet generation rate). Tr(n) represents the time the packet arrives, and it is the responsible of the presence or not of the jitter in the connection: 3 different simulations are done according to the way of calculation of Ts(n):

- No delay: in every packet Tr(n) = Ts(n). This is the same case as the one in the section 5.1.1 No delay simulation, so the same result is expected: jitter should be 0.
- Constant delay: Tr(n) = Ts(n) + constant delay value. As explained in section 5.1.1 the jitter must be 0 as well.
- Random delay: Tr(n) = Ts(n) + random delay value. In this case, the delay is calculated by multiplying a delay coefficient by a random value in the range (0, 1). Jitter is present and should remain within two times the maximum delay.

All these connections are captured by running the Python code, as it generates the .pcap file in the destination that is indicated.

5.2.1 No delay and constant delay

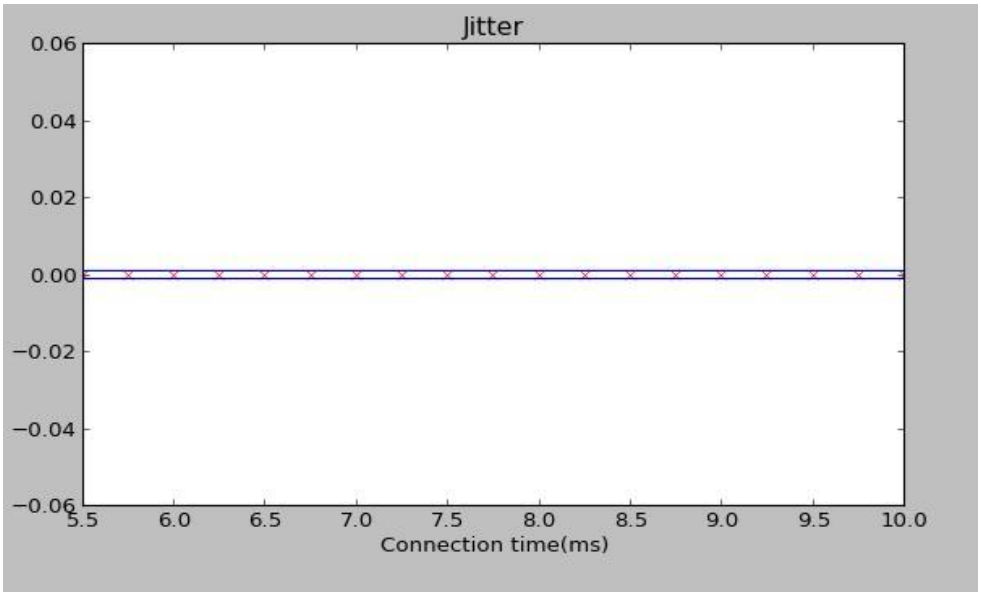

Figure 15: No jitter generated by Python code

The reason why "No jitter" case brings together the first two configurations is because in this case, Tr(n) value is being set manually by the programmer, so setting it equal to sending time (no delay), or equal to sending time plus a constant value(constant delay), leads to the same result. Figure 15 shows how the jitter is absolutely 0, not even propagation delay is registered.

5.2.2 Random delay

In this case the Tr(n) value is calculated by increasing
Tr(n) = Ts(n) + 20 (ms) *random(0, 1), and so, the jitter remains within ±20 ms. Figure 16 shows this behavior over a 100ms simulation.

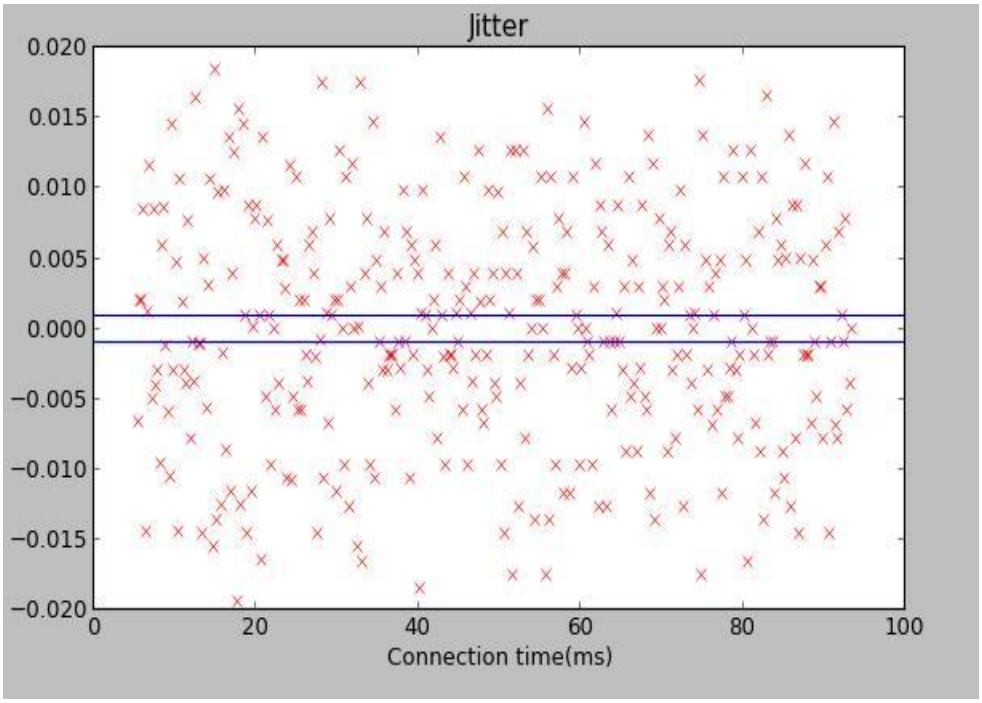

Figure 16: Random delay generated by Python code

We have generated and sent packets at time Ts(n), put the t(n) values inside the packet (using the Equation 2), added the delay to Ts(n) according to the case it was being run and, finally, recorded this value as Tr(n). It has been simulated a real analysis at the receiver side where only Tr(n) and t(n) are accessible, and Ts(n) has to be derived from t(n) assuming certain error, and the results shown match the reality: the study of the jitter shows the expected results at every case, so the algorithm has successfully passed these tests developed through section 5.2.

5.3 Ns-3

The ns-3 network simulator is a very powerful tool used to simulate networks and transmission of packets in them. In this thesis work this tool is used to simulate simple connections, using TCP/IP protocols over Ethernet.

First of all the nodes are created and they are connected by a CSMA channel, in order to add the Ethernet headers. At this point the delay of the channel is fixed. After that, the TCP/IP protocols are installed in all nodes, with the help of TCP sockets. This may require special attention that instead of ns-3 default TCP protocol, *NscTcpL4Protocol* is installed in some of the nodes, in order to activate the TCP timestamp option in the connection. Finally, the Application layer is defined: in one node an OnOff client is installed (to transmit whenever it has something to transmit) and in the other node a PacketSink server is set, that receives the information sent from the client side.

Once the connection is configured, the data of the transmission is saved in different files: a .pcap file with all the packets that are sent, a XML FlowMonitor, that register all the activity in both nodes, and the server prints in the terminal every packet that is received.

When using the ns-3 simulations we wanted to generate, the same cases as with the previous two tools. Mainly a first case where the delay is constant over time, and a second case where the delay fluctuates over time, so jitter is present in the channel. However, when working with this simulator, the way of adding delay to the channel is slightly different; once the delay is set as a channel attribute, is not trivial to make it change over time. And the case where this delay is a random variable was not possible to be implemented. The solution achieved is the following: we create a CSMA channel with a certain number of nodes; only one of these nodes is connected to a single node by a point-to-point channel, and finally this node is connected to another single node by another point-to-point channel. Two cases are then generated: constant delay model and random delay model.

5.3.1 Constant delay model

In this case only two nodes are connected to the Ethernet link: Node A is the OnOff client, while Node B is connected to Node C by a point-to-point channel. As only one client tries to access the server (Node D), no traffic or queues are present, so the delay suffered by the packets should be the sum of all the propagation and transmission delays in all the 3 channels. No jitter should be present as all packets experience the same delay.

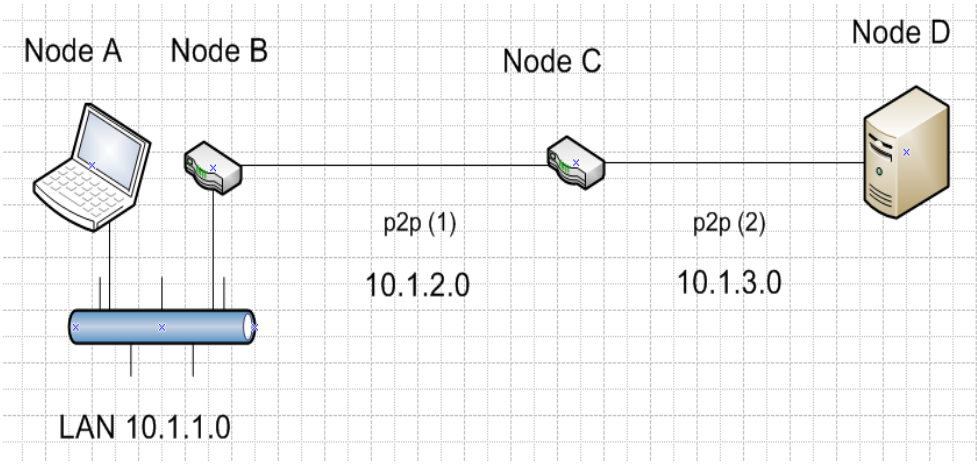

Figure 17: Constant delay ns-3 model

The figure below shows the results when running the ns-3 configuration by the Python code.

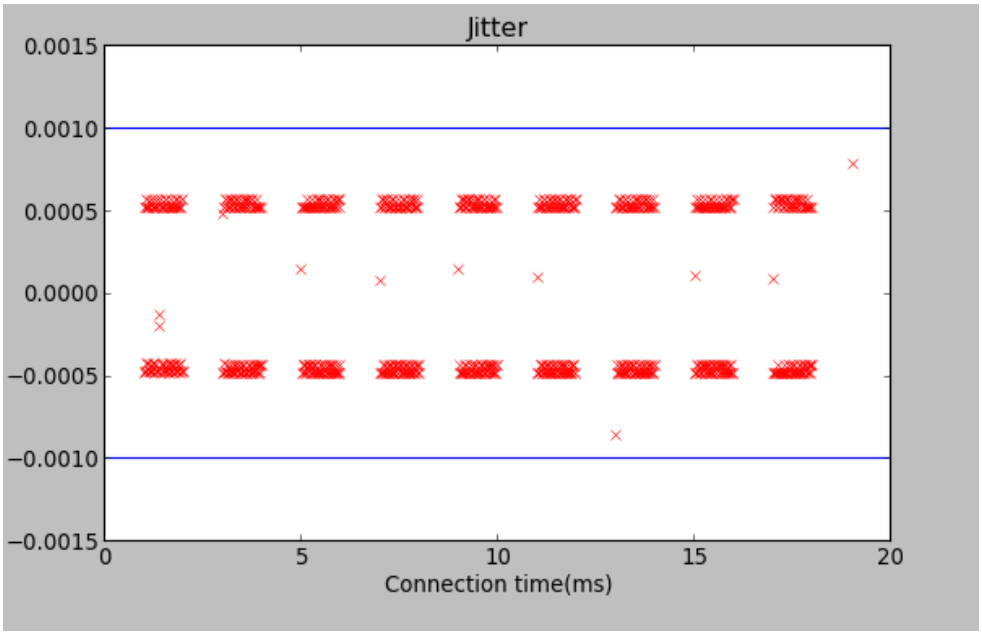

Figure 18: Constant delay ns-3

The client is accessing a server, and no more traffic is present in the network. That is why the delay is fixed and the same for all the packets, so no jitter appears in this simulation. As explained in section 5.1.1, the fluctuations of the jitter within the two blue lines (that represent ± 1/F) cannot be distinguished from noise, due to the floor function that introduces some error into the calculations.

5.3.2 Random delay model

This configuration is very similar to the previous one. However the Ethernet link contains 7 nodes instead of only 2. The purpose of this configuration is to install an OnOff client in every node connected to the CSMA, whereas the server is located at Node D. During the first 8 seconds all of the nodes are active, trying to reach the server, but as only Node B can communicate with the server, a queue builds up, so the packets sent from Node A should be delayed. As delay is random (or at least not constant), jitter must be present in this simulation.

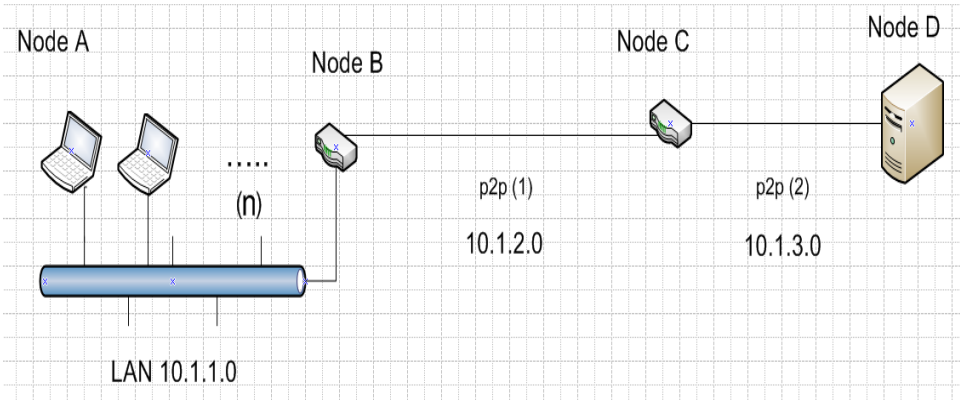

Figure 19: Random delay ns-3 model

Running this second scenario, where several clients connected to a CSMA device try to access the server, the following plot is obtained:

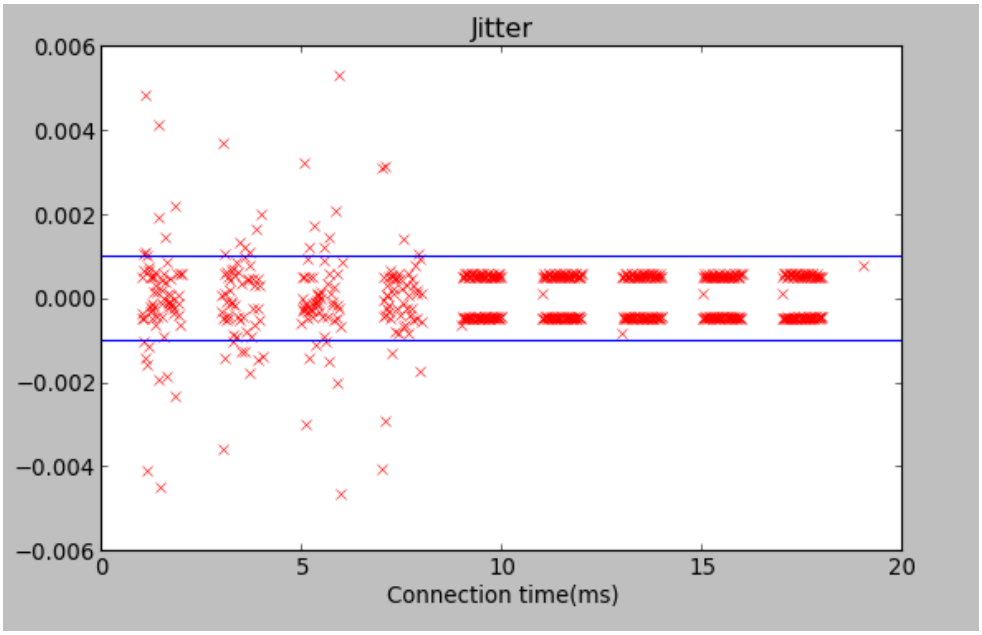

Figure 20: Random delay ns-3

The plot in the image shows the .pcap file at one of the nodes connected to the CSMA. This node is transmitting during 20 seconds, while the other 6 nodes transmit only during the first 8. That means that during the first 8 seconds of the connection, collisions and traffic appear in the network, and so random delays. All these cause jitter as shown in the Figure 20. Once the rest of the clients are stopped, the jitter is again non-existent, as in section 5.3.1. Regarding both ns-3 tests, the results validate that the algorithm is able to detect jitter when present.

CHAPTER 6

Real data analysis

The code implemented and validated through chapters 4 and 5 is going to be used to test the QoS in terms of latency on the Internet. First of all, we are going to confirm that the guard time established in chapter 4 is enough to give an F calculation close to its nominal value. Then we are going to run the algorithm over different capture files depending on which case wants to be analyzed. And the last part will be analyzing the results in order to give conclusions on how the different cases affect the QoS.

The QoS is studied in terms of **jitter standard deviation** and **maximum queuing time.** The reason why the jitter is studied in terms of its standard deviation and the queue size not is that the jitter is calculated between every pair of packets, and so an unique value must be considered for every single connection; as it is used to study the fluctuation of the arriving packets, the standard deviation is the factor chosen to show that dispersion (from the mean value), while the queue algorithm only delivers the maximum queue for every connection; calculating max queue tells that on the path the packets in that connection travel, there is a possibility to build up that large delays. But maximum queue size algorithm cannot tell the difference, for example, between a connection with 9999 packets with 0ms jitter and 1 packet with 1000ms and a connection with 9999 packets with 999ms jitter and 1 packet with 1000ms. There needs to be something that summarizes the jitter, which is the jitter standard deviation. Measurement results are collected and grouped into three distinct cases which we find interesting to observe:

- Sending from a **wired or** a **wireless** network, to test if the type of network affects the QoS.
- Accessing the **same server,** in order to see if there are certain servers that show high latency when accessing them.
- The **time** when the connection was maintained; this case wants to show if there are some “rush-hours” when the network is congested and so the QoS of the connection will be very bad.

These three cases are selected focusing on the influence that a certain factor (type of network, server accessed or time the connection is established) can have on the QoS of the connection. We found them interesting because defining how they affect the latency, the ones that produce bigger latency can be avoided if possible (if we can choose between a wireless and a wired network, or if we can choose the moment to establish a connection for example), and so the QoS can easily be improved.

The influence of sending from a wired or a wireless network, and the correlation of these two measurements when sending from the same host can be studied. It is also interesting to show the evolution of the jitter standard deviation vs. the connection time. This shows if at any moment of the access, there are a lot of jitter spots with very big values. To achieve this, the jitter values bigger than 1 s are recorded, considered as rare cases, together with the host and time this happened. At this point, time periods with big jitter in the network can be detected, as well as hosts that show high latency when accessing them.

6.1 Procedure details

First of all, the procedure used to calculate the F factor is checked to confirm that it works when running over these new .pcap files. The figure below shows the CDF of both packet and time margins explained in section 4.1.

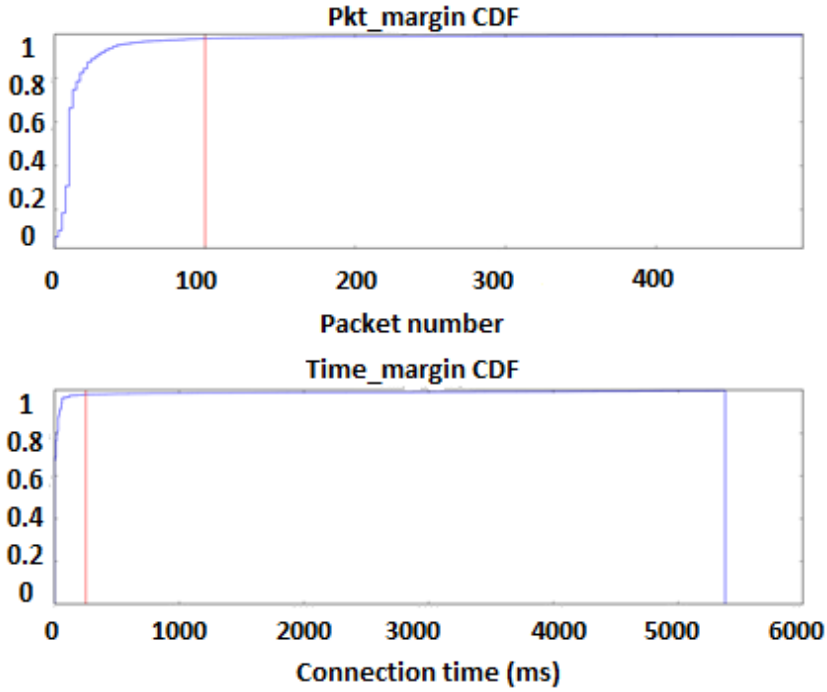

Figure 21: CDF of packet and time margins in capture-20130404-123423.pcap (zoomed)

It is seen that the packet margin covers up to 99.5% of the connections present in this file, while the time margin does in 98.5 %, so it works even better than in the previous case where these margins were determined. Once this issue is double checked, the data in the files are analyzed according to the three cases mentioned above; two big .pcap files are used to gather the results of the analysis:

- capture-20130404-123423.pcap to study the influence of accessing a wired or a wireless network . This file's details has been explained in section 4.2
- parsed_capture-2013.pcap.gz: collection of text files representing a generic Internet traffic scenario, whose details are also explained in section 4.2

Regarding the jitter standard deviation; the algorithm calculates the jitter between every consecutive pair of packets in every connection both at the sender and the receiver. For every connection, the standard deviation of these jitter values is stored together with its IP address. At the end, a series

of IP addresses are obtained with their respective jitter standard deviation values. The interesting ones are those that are accessed more than once, as they can be studied to check whether they resemble each other in every access. Like in the jitter standard deviation calculation, the maximum queuing time value is stored together with the IP address of the server accessed.

6.1.1 Results when sending from a wired or wireless network

The capture-20130404-123423.pcap file with all the data collected when sending from different hosts have been obtained by the following scheme:

Figure 22: Collection of data scheme

One of the most interesting features when analyzing this file is comparing the differences between the jitter and the queuing suffered by the packets when sending from a host which is on a wired or a wireless network. Thanks to the middlebox provided by the company, two networks are used.

- 172.21.20.0/24 --> Wired
- 172.21.21.0/24 --> Wireless

These are IP addresses for two private networks and they are in the middlebox, while the caption is being made from the office. So every attempt to connect to Internet is forced to go through one of these two ranges of IP addresses. Running the algorithm explained in chapter 4, the following plots are obtained:

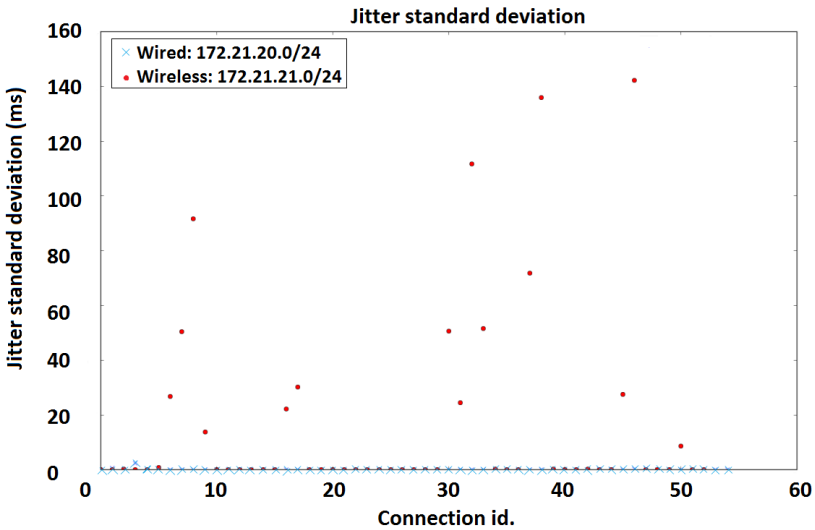

Figure 23: Jitter standard deviation in wired and wireless networks

Figure 23 shows the jitter standard deviation values when sending from the wired and wireless networks. It is clearly seen that the spots related to the wireless host (red circles) show a bigger value than the ones related to the wired network (blue crosses). As it was expected, it can be concluded that the jitter is more harmful when sending from a host that is in a wireless networks than when is in a wired one, as it deviates from the average more and so it is more spread.

Figure 24 and Figure 25 show the jitter standard deviation histogram regarding both hosts (with the x axis in linear scale). While the wired network shows a jitter standard deviation around 0, with values up to 2.5 milliseconds, the wireless one is more spread, with some values up to 150 milliseconds.

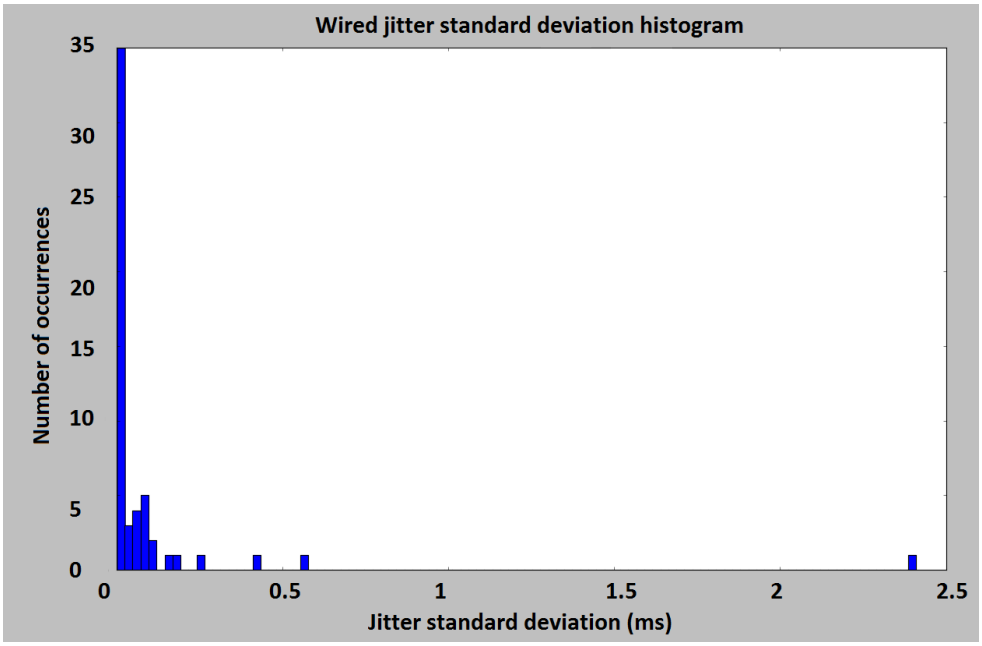

Figure 24: Wired network's jitter standard deviation histogram

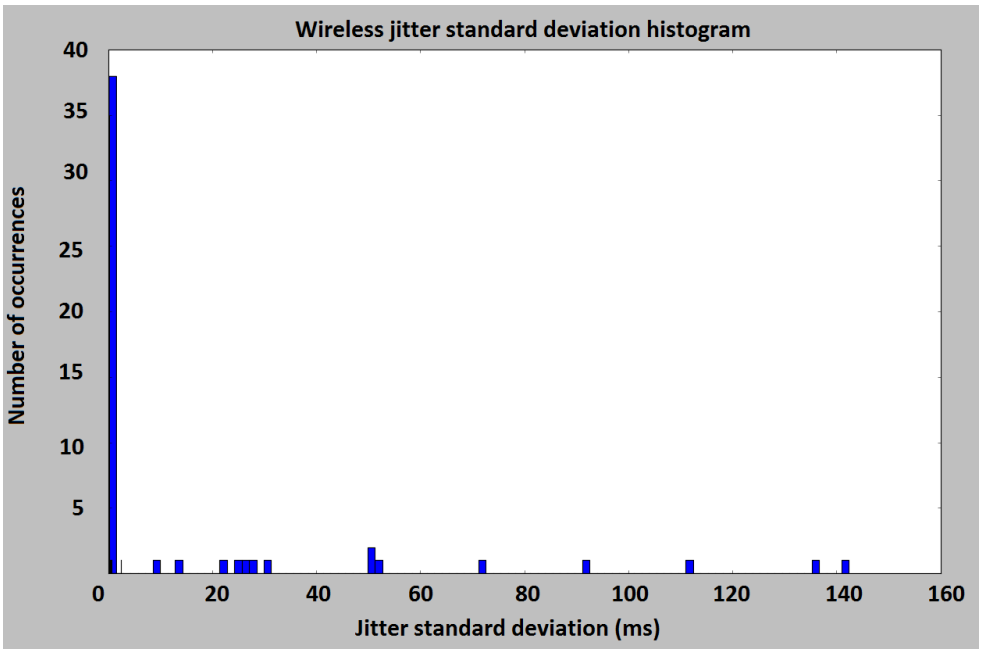

Figure 25: Wireless network's jitter standard deviation histogram

This confirms that the jitter fluctuates more when using a wireless host than when using a wired one.

Like the jitter standard deviation, the maximum queuing time is expected to be bigger in wireless than in wired hosts. The three plots below show the results:

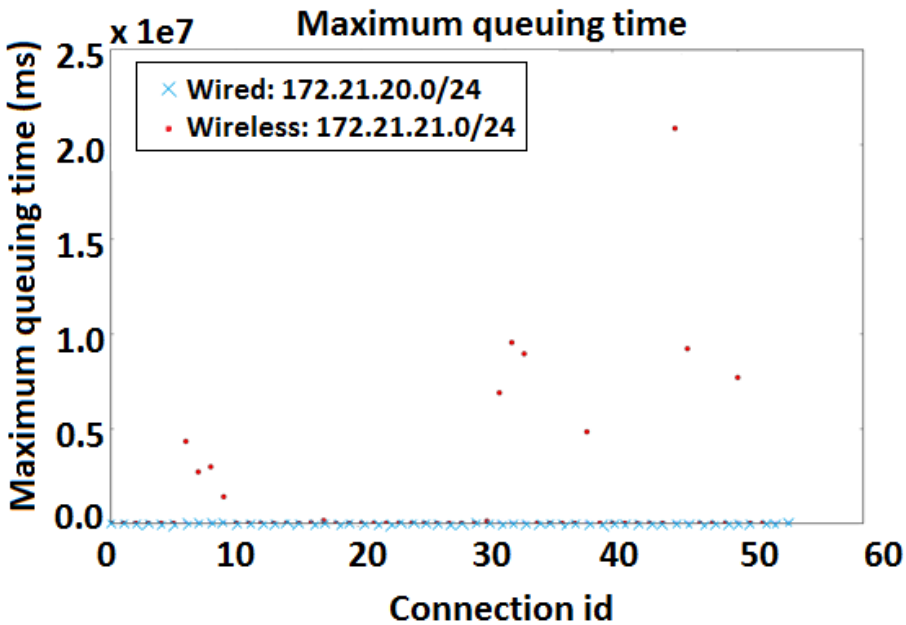

Figure 26: Maximum queuing time in wired and wireless networks

Figure 26 shows the spots regarding the maximum queue experienced by a packet when sending from a wired or a wireless host. The information that can be extracted is that when using a wireless network (red circles) the queues are bigger and so the delays, comparing to a wired network (blue crosses).

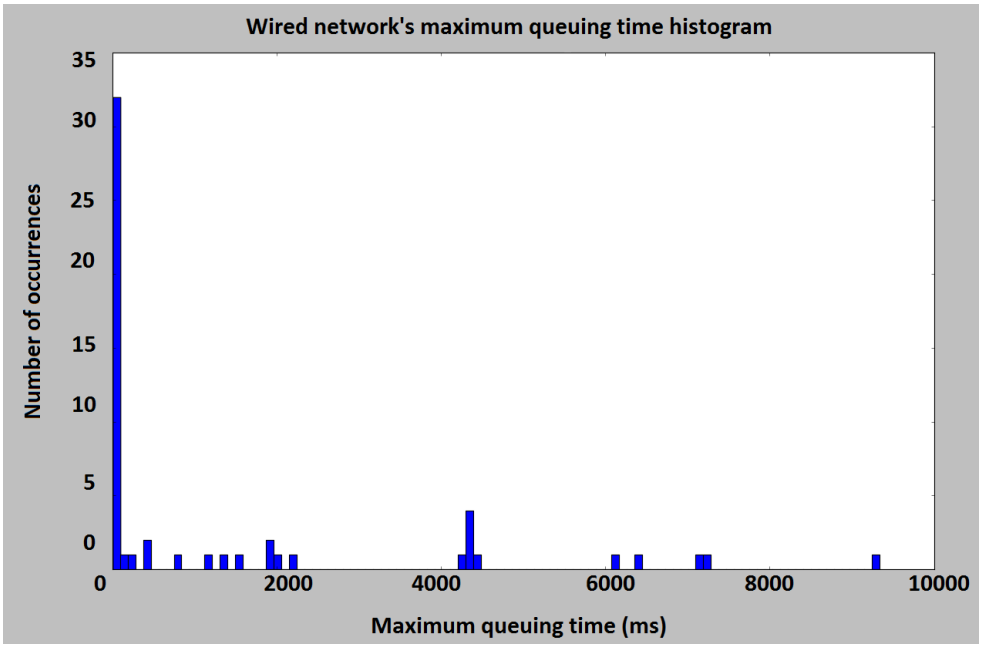

Figure 27: Wired network's maximum queuing time histogram

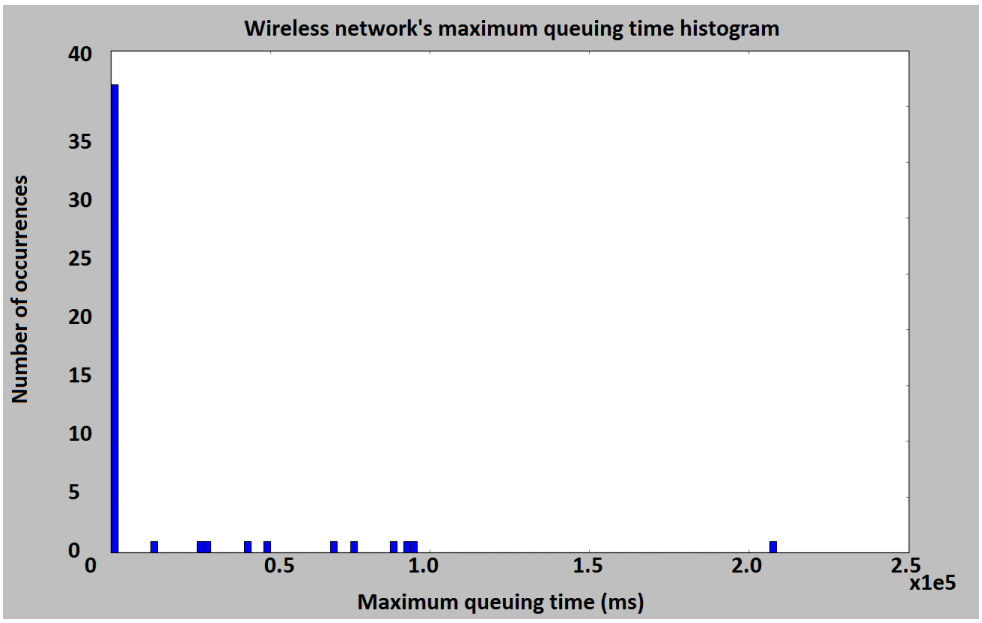

Figure 28: Wireless network's maximum queuing time histogram

Figure 27 shows that the queuing when sending from the wired host is mostly around 0, with peaks up to 9 seconds, while Figure 28 reveals that in some cases, the queuing in the wireless server can be huge (100's of seconds).

As expected, the maximum queuing analysis tells that the QoS is worse in wireless links than in wired ones, as the jitter standard deviation did. These two histograms show as well that the maximum queuing time is more spread than the jitter standard deviation and so more difficult to determine.

6.1.2 Results when accessing a certain server

In this section the next case is studied: the server accessed. The aim is to determine if there is a chance that depending which server is being accessed, the latency can be bigger. At this point of the analysis, the collection of parsed_capture-2013*.pcap.gz data files is used. As it was expressed in section 4.2, this represents a real Internet scenario due to the amount of selected data. The way this study is done is the following: first of all, detect all the spots that correspond to a jitter bigger than 1s. For each of these spots, the server accessed when generated is recorded. Finally a histogram shows the number of big jitter occurrences/number of packets per server.

The amount of data present in this collection of data files makes the algorithm running for long periods of time. That is why all the 39 .pcap.gz files are split in 6 groups in order to make the simulation shorter and more efficient. The figures below are 2 examples of two of these 6 groups:

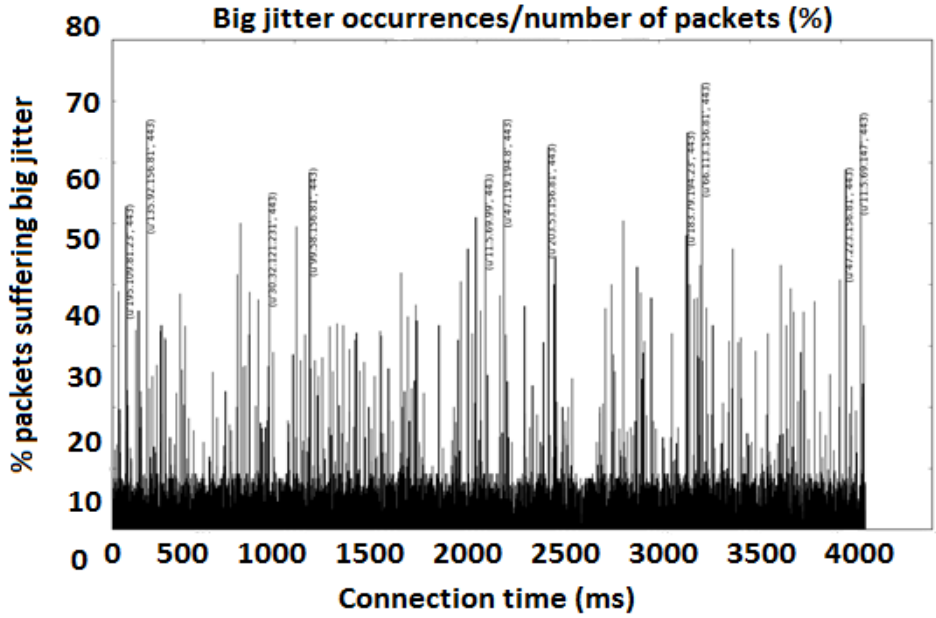

Figure 29: Big jitter occurrences per server (1)

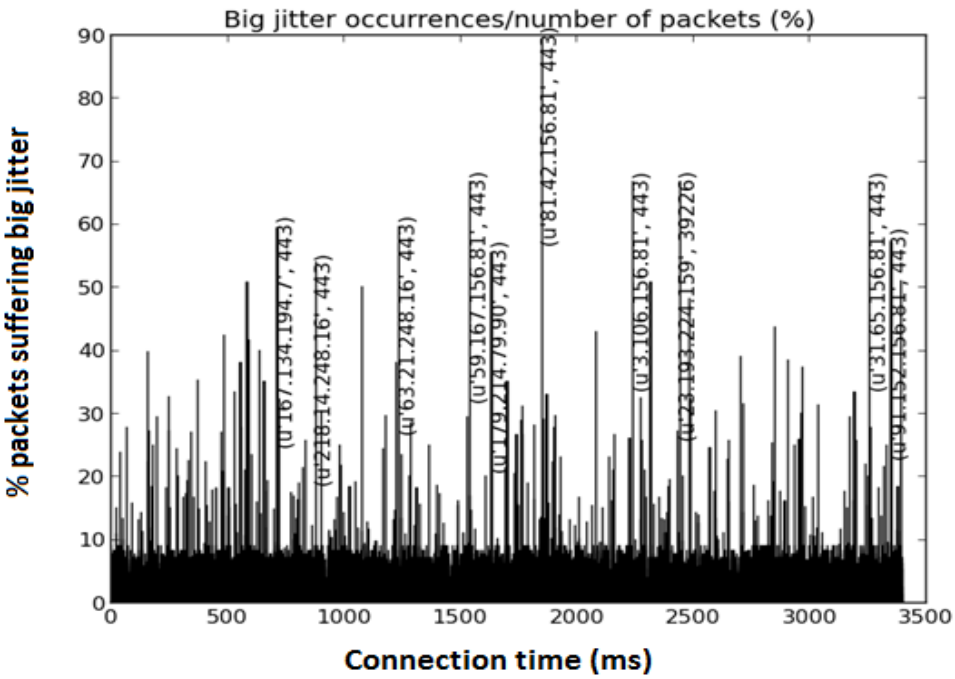

Figure 30: Big jitter occurrences per server (2)

It is clearly detected that some servers show a big percentage (even higher than 70%) of packets suffering from big jitter, while others do not show jitter at all. This feature is studied more in detail by choosing a server that shows big latency and another one that shows a small one.

- **59.185.143.0/24** → network address range accessed 23 times and 7213 packets are exchanged. It shows 77 % of packets suffering from big jitter.
- **170.126.59.0/24** → network address range accessed 148 times exchanging a total amount of 21.123 packets. It shows 3 % of packets suffering from big jitter.

The jitter standard deviation and the maximum queue are taken as the parameters that determine the QoS of the connection.

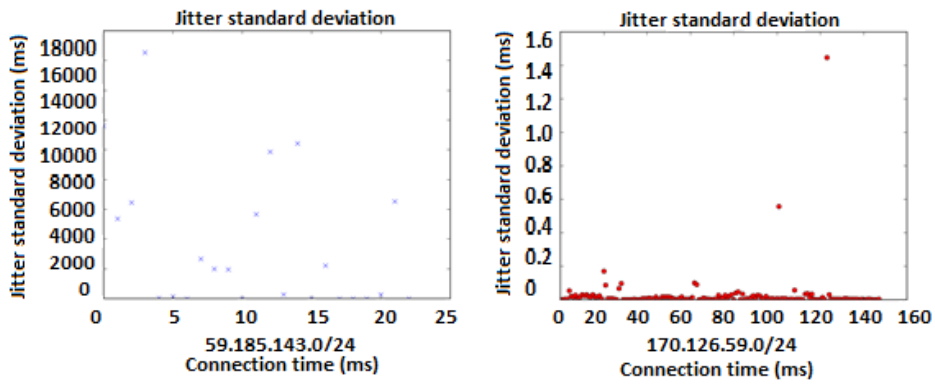

Figure 31: Jitter standard deviation

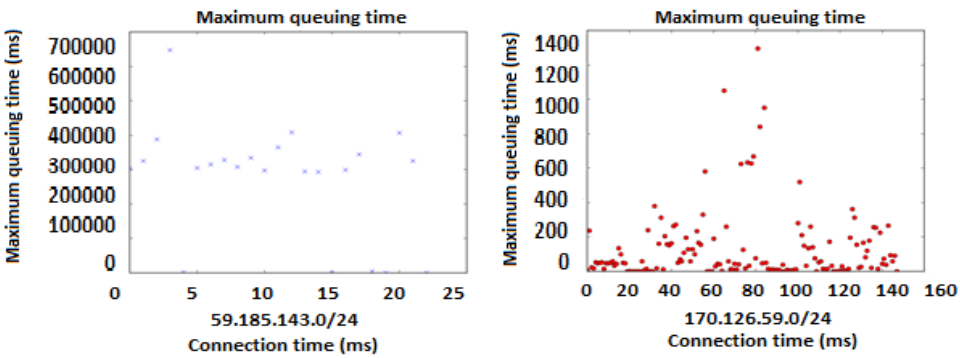

Figure 32: Maximum queue time

In all the 4 plots above, there is a server whose jitter deviation and queue size values are very low and concentrated (the right plots → 170.126.59.0/24), while the left ones (corresponding to 59.185.143.0/24) show very spread values and in certain cases extremely big ones. It is very interesting to note that the server 170.126.59.0/24 exchanges almost 4 times more packets than server 59.185.143.0/24; nevertheless in a smaller amount of packets, almost 80 % of them are showing a big jitter. Our conclusion is that, when accessing certain servers, latency can be more likely to occur. In other words, **the latency experienced in a connection, as well as the QoS, can be seriously impacted when accessing a certain server**; a well-managed server handles heavy traffic better than the other, which is a badly-managed server that cannot even handle light traffic. By identifying these badly-managed servers, and if possible avoiding them, the latency can be improved.

6.1.3 Results for different connection times

Finally, the last case to be taken into consideration is the time the connection was established. It can happen that the network is overloaded during certain times, and the QoS in the connections established during these busy periods can be seriously impacted.

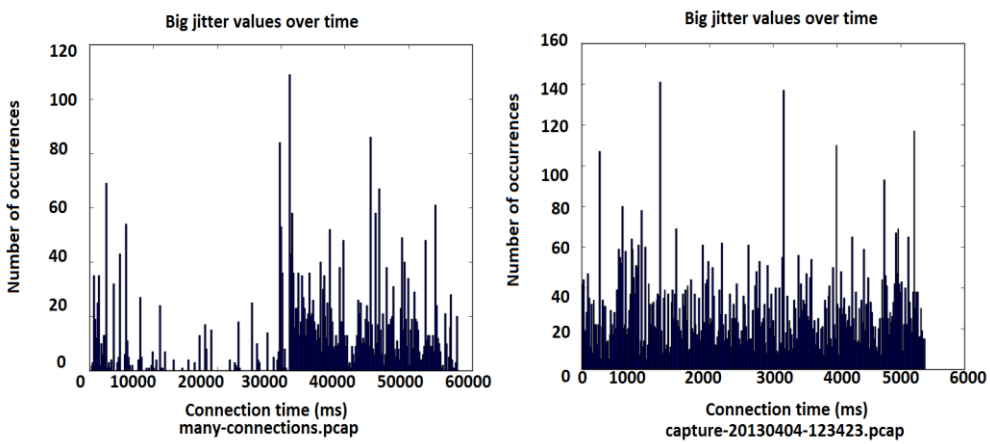

Figure 33: Big jitter values over connection time (1)

Figure 33 shows the histogram of two different .pcap files: to the left the study of many-connections.pcap, and to the right the study of capture-20130404-123423.pcap. The x-axis represents the time since the connection started, and the y-axis the number of packets that experienced big jitter at this moment. Internet latency is very random, and so trying to elaborate a model is difficult. The plot to the right shows no trend in that connection, so no action can be taken. However there are some connections where during a certain period, the number of packets suffering high jitter is bigger. The plot to the left shows how between 30 – 45 seconds, the network is congested, and so a big amount of packets are facing big jitter.

The same behavior can be seen by looking at the next plot: it represents the time histogram of one of the 6 groups used to analyze the file parsed_capture-2013*.pcap.gz:

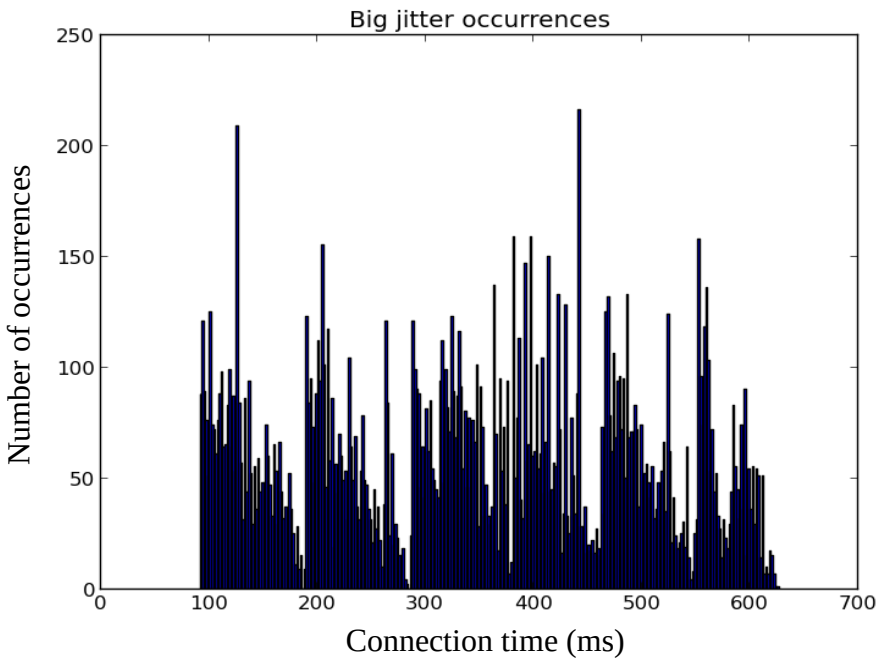

Figure 34: Big jitter values over connection time (2)

In this case the big jitter occurrences are kind of periodic (there are lapses of 100 ms with big jitter and suddenly it becomes very small).

The conclusion from this case is that **there are some periods where the network is congested, so the QoS of the connection is impacted** depending on when it takes place. Avoiding rush hour can help the latency to be lower. At the end is a matter of the user if the connection must be established in this certain moment (time priority) or if it is preferred to wait certain time in order to get a better QoS (quality priority).

The three different cases have shown how there are situations that are very likely to produce a high latency in the network. Thanks to the algorithm developed in this thesis, these situations have been analyzed and defined, and in certain cases are easy to avoid. The information provided by the algorithm may be used in future network improvements in order to take active solutions that may focus on the avoidance of the high-latency cases.

CHAPTER 7

Conclusions and future work

The main focus of this thesis has been to develop an algorithm taking advantage of the TCP timestamps options in order to study the behavior of latency in Internet. The data used to make the study have been collected by a middlebox placed at Procera Networks, the company collaborating with us in this thesis. These trace files contain interesting data and a representative selection of the Internet traffic in order to make a study as accurate as possible. The algorithm developed also derives some statistics based on the collected data.

In order to validate the algorithm and its implementation in Python, several test connections are generated in three different ways: using Netem Linux tool and sniffing one computer when accessing a HTTP server in another PC, running a Python code that fills in the gaps of the different values used in the .pcap files, and finally, the more interesting because of the power of the tool has been the ns-3 simulator, which is highly extended nowadays in network analyzing. Once the Python code has been validated and works as expected, real data is analyzed in order to find factors that have a negative impact on the QoS in terms of latency of a connection.

The first observation we made was that latency is higher in wireless than wired networks. So the traffic sensitive to the latency (for example, VoIP) experiences a poorer QoS when sending from wireless networks (in general terms). Even when accessing two different wired servers, some of them show a higher latency. Not only the type of server (wireless/wired) is relevant, also the specific server that is being accessed. And the last factor affecting latency and bufferbloating in Internet is the moment the transmission takes place. During the time the connection lasts, there are some periods where the packets experience bigger queuing times and jitter.

This new algorithm used to study the latency and bufferbloating in Internet has tried to provide more accurate results than previous ones thanks to TCP timestamps option. Nevertheless, latency is a difficult parameter to avoid

due to the randomness of traffic in Internet. In order to avoid the big latency mainly in interactive applications, some studies and models can be developed in the future; these models should focus on trying to detect which servers are more likely to show a big latency when accessing them, and on detecting the rush-hour in order to avoid periods where the network is congested and so the latency will be much bigger.

Our work has been developed by analyzing previously recorded traffic traces stored in .pcap files. This offers a good approximation, but also limits a bit the usage of the algorithm; if the algorithm is run in live networks, all these factors can be calculated live as well and so determine when it is a good moment to establish the connection. Studying live networks also offers a bigger range of connections to study, but if the .pcap file is smartly stored, this disadvantage can be reduced.

The trace files used during the work have been collected with the help of a middlebox, which uses DPI technology. This feature allows a big advantage, as the traffic recorded can be easily selected to show a wide range of different connections, while keeping the confidentiality. However, it can be used to provide better QoS; by analyzing all the traffic coming through the middlebox, and setting priorities, the most urgent traffic can avoid latency. In this case, the most latency-sensitive traffic can be marked as high priority (VoIP, streaming…), while the less interactive applications' traffic can be set as low-priority (downloads, web-browsing…). DPI is an important way to reduce latency in some connections, and so it must be utilized.

As all this Master thesis is focused on a passive analysis in order to study the current scenario, the results shown can be taken into consideration in order to develop active procedures that can improve the latency and QoS in Internet.

References

[1] J. Gettys, V. Jacobson, N. Weaver and V. Cerf, "BufferBloat: What's wrong with the Internet," *Communications of the ACM,* vol. 55, no. 2, pp. 40-47, February 2012.

[2] J. Gettys, "Bufferbloat.net," January 2012. [Online]. Available: http://www.bufferbloat.net/projects/bloat/wiki/Introduction. [Accessed February 2013].

[3] J. Gettys, "Dark buffers in the Internet," Magazine Communications of the ACM, Volume 55 Issue 1, Pages 57-65, ISSN: 0001-0782, EISSN: 1557-7317 , New York, 2012.

[4] Y. Gong, D. Rossi and C. Testa, "Interaction or Interference: can AQM and Low Priority Congestion Control succesfully collaborate?," Proceedings of the 2012 ACM conference on CoNEXT student workshop, New York, 2012.

[5] H. Balakrishnan and V. N. Padmanabhan, "How Network Asymmetry Affects TCP," IEEE Communications Magazine, 2001.

[6] L. Kalampoukas, A. Varma and K. K. Ramakrishnan, "Two-Way TCP Traffic over Rate Controlled Channels: Effects and Analysis," IEEE/ACM Transactions on Networking, Vol. 6, No. 6,, 1998.

[7] H. Balakrishnan, V. N. Padmanabhan, G. Fairhurst and M. Sooriyabandara, "RFC3449 - TCP Performance Implications of Network Path Asymmetry," RFC Editor, 2002.

[8] J. Dangaard, "Master thesis: Optimization of TCP/IP Traffic Across Shared ADSL," University of Copenhaguen, Denmark, Copenhaguen, 2005.

[9] S. Cheshire, "It's the Latency, Stupid (Part 1)," Stanford University, Stanford, 1996.

[10] Celtic-Plus, "IP Network Monitoring for Quality of Service Intelligent Support," October 2012. [Online]. Available: http://projects.celtic-initiative.org/ipnqsis/. [Accessed April 2013].

[11] Y. D. C. S. Vinton Cerf, "RFC 675 - Specification of Internet Transmission Control Program," RFC Editor, 1974.

[12] V. Jacobson, B. Braden and D. Borman, "RFC 1323 - TCP extensions for High Performance," RFC Editor, 1992.

[13] K. Fall and S. Floyd, "Simulation-based comparisons of Tahoe, Reno and SACK TCP," ACM SIGCOMM Computer Communication

Review , Volume 26, Issue 3, New York, July 1996.

[14] S. Floyd, T. Henderson and A. Gurtov, "RFC 3782 - The NewReno Modification to TCP's Fast Recovery Algorithm," RFC Editor, April 2004.

[15] K. Tan, J. Song, Q. Zhang and M. Sridharan, "Compound TCP: A Scalable and TCP-Friendly Congestion Control for High-speed Networks," 4th International workshop on Protocols for Fast Long-Distance Networks (PFLDNet), 2006.

[16] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," IEEE/ACM Transactions on Networking, 1993.

[17] D. L. Black, S. Floyd and K. K. Ramakrishnan, "RFC 3168 - The Addition of Explicit Congestion Notification (ECN) to IP," RFC Editor, 2001.

[18] P. Karn and C. Partridge, "Improving Round-Trip Time Estimates in Reliable Transport Protocols," SIGCOMM '87: Proceedings of the ACM workshop on Frontiers in computer communications technology, New York, 1987.

[19] C. Chirichella, D. Rossi and C. Testa, "Remotely gauging upstream bufferbloat delays," *Proceedings of the 14th international conference on Passive and Active Measurement*, pp. 250-252, 2013.

[20] M. Allman, "Comments on Bufferbloat," ACM SIGCOMM Computer Communication Review, Berkeley, CA, USA, 2013.

[21] T. Porter, "Symantec," 19 October 2010. [Online]. Available: http://www.symantec.com/connect/articles/perils-deep-packet-inspection. [Accessed February 2013].

[22] Procera Networks, "Procera Networks," [Online]. Available: http://www.proceranetworks.com/products-overview.html. [Accessed February 2013].

[23] M. Donegan, "Spanish Mobile Challenger Wields DPI," *www.lightreading.com*, p. 1, 08 March 2011.

[24] M. Donegan, "TeliaSonera Develops New Mobile Data Model," *www.ligthreading.com*, p. 1, 28 July 2010.

Appendix 1: Python libraries used

- iPython: Python shell that makes much easier programming in Python (tab help, more graphical interface…)
- Pcapy: reading into .pcap files obtained for example from wireshark
- Impacket: reading into the packets present in the .pcap file, allowing access to headers, the different protocol fields…
- Matplotlib: graphical tools library that resembles Matlab programming.
- IPy: handle IP addresses in XXX.XXX.XXX.XXX/XX format