POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Implementation of an Available Bandwidth estimator for WAN networks

Supervisor
Prof. Fulvio Giovanni Ottavio Risso
Ing. Francesco Ciaccia

Candidate
Luigi Napoleone Capasso

April 2020

Summary

This Thesis Project has been developed at Clevernet, a company which specializes in applied research of computer networking, and whose main aim is to improve the Internet by providing secure data-in-motion. In particular, this Research Project was focused on avoiding the so called self-induced congestion, which consists of buffering and packet loss which occur in any of intermediate routers between two end-points. To pursue that, it was necessary to act on the Congestion Control algorithm used in TCP. It is an end-to-end transport layer protocol and it controls and monitors the network flow and the sending rate through two main mechanisms: Flow Control and Congestion Control. The former is an end-to-end mechanism used to avoid having a sender send data too fast for the receiver, which has to receive and process it reliably. The latter aims to provide reliability to the connection between two hosts, by preventing a node from overwhelming the network. However, Congestion Control algorithms still represent a limitation in exploitation of network resources, basing their work only on indications of lost packets as a signal to slow down. Loss-based Congestion Control is problematic in today's diverse computer network landscape. [1] suggests a way to mitigate these issues, which improves end-to-end TCP performance, latency and fairness. The basic idea is to modify the Receive Window (RcWnd) belonging to the receiver so that any TCP communication is way faster and, possibly, without delays and packet loss. Estimating available network resources is also fundamental when adapting the sending rate both at the application and transport layer and for this reason, it is imperative to estimate a path's Available Bandwidth in real-time. In order to do so, a statistical method based on the inter-packet arrival time analysis of TCP acknowledgements is proposed. Additionally, the round-trip time is estimated between the sender and the receiver of the connection.

Acknowledgements

I would like to thank my parents, who enlightened my days with their smiles and their voices. They are my point of reference, they financially and emotionally supported me. They gave me the opportunity to study in Politecnico, which is one of the most important University in Italy and expecially to go abroad to work on my thesis. They both were in my heart during my free times, my classes, my exams, my nights. They gave me everything I needed, without questions.

In particular my father, who always wished me "good luck and be quiet" before all my exams. He gave me a lot of recommendations about this my path, and I am so proud to have him in my life and to count on him whenever I am in trouble.

My mother, who always respected my choices and believed in my capabilities and wiped my tears for me when I had to face difficult moments. Every time I felt bad or I was sad she was ready to give me help.

My brother who gave me some nice tips about university life.

Alisa, my girlfriend, who I met during my first year of Master's Degree. She is one of the most formidable girl I know. She is my best friend and she taught me that I have to be always positive and smile to the life. Her smile and her love made me achieve good marks and improve my self-esteem.

My all colleagues who I prefer calling friends who I met in these last five years, with whom I spent great time and moments to share. I will never forget the classes together and all the funny things we have done at University and in our free time.

Finally, I would like to thank all the employees working at Clevernet for their availability. In particular my supervisor Francesco Ciaccia, who followed and supported my whole work and Ivan Romero, with whom I spent a lot of good time.

Contents

Chapter 1

Purpose of the Thesis

This Research Project was focused on mitigating the so called self-induced congestion. Such phenomenon consists of buffering and packet loss which occur in any of the intermediate routers along the path between a connection's end-points due to the significant amount of traffic emission patterns at the source. To pursue that, it was necessary to act on the Congestion Control algorithm, which is one of the most important algorithms in the world of TCP. This algorithm controls and monitors the network and provides reliability to a TCP connection between two hosts, by preventing a node from overwhelming the network. This should limit congestion phenomena inside the network, which can cause packets loss and significant delays. This is done by dynamically setting the amount of data to send to the receiver, in order to provide a good service; moreover, this monitoring is done on the end-points of the network and not inside the network itself. However, Congestion Control algorithms still represent a limitation in exploitation of network resources, especially those involving a great number of concurrent requests to an host. They fundamentally base their work only on the indication of lost packets as a signal to decrease the packet rate. But loss-based Congestion Control, which is widely used nowadays, is problematic in today's diverse networking solutions. For example, in shallow buffers, packet loss happens before congestion. With today's high-speed links that use switches with shallow buffers, loss-based Congestion Control can result in abysmal throughput due to the system's overreaction, decreasing the sending rate by half upon packet loss. In deep buffers, congestion happens before packet loss. At the edge of today's Internet, loss-based congestion control causes the so called "bufferbloat" problem, by repeatedly filling the deep buffers in many last-mile links and causing seconds of needless queuing delay [4]. Cubic and BBR algorithms are widely used nowadays; the former is a loss-based Congestion Control algorithm for Linux kernel, the latter is a model-based algorithm, and works better than the former. However the former acts too aggressively, because it aims to lose packets, the latter is not very precise when there are many TCP transfers in parallel. Thus, the main purpose of this Thesis is to

improve TCP performance and, in doing so, reduce self-induced congestion caused by loss-based Congestion Control algorithms. We wanted to develop a lossless throttling mechanism, that is, decreasing the rate of all TCP flows without packet loss. With this goal in mind, we had the idea to develop a new TCP Congestion Control algorithm, which is still loss-based, but reacts to packet loss and does not need it to understand that the network is not working properly. To do that, we need to enforce an upper bound to the sending rate of the sender, exploiting TCP flow control. This is achieved by reducing the advertised window of the receiver of in-flight acknowledgement packets and can be later done by a dynamic controller deployed on the edge routers of the network, as the edge deployment vantage point gives the router visibility on all the flows being generated in the Internet access link. This allows a fair partitioning of the edge router resources in terms of bandwidth. For this purpose, an accurate estimation of the Available Bandwidth was needed, and scientific studies led Clevernet to the following formula to compute it:

$$\text{IP rate} = \frac{Ack_i - Ack_{i-1}}{Ts_i - Ts_{i-1}}$$

It is equal to the difference between two acknowledged TCP packets (in terms of Bytes) divided by the difference between the two packets' acknowledgement timestamps.

Chapter 2

TCP: a short overview

2.1 What is TCP

The Transmission Control Protocol (TCP) is one of the most important protocols belonging to the Internet protocol suite, which provides reliability and error-checked delivery of bytes exchanged between applications running on hosts connected through an IP network [3].

2.2 Main features

TCP provides several features which make it unique and interesting:

- Connection oriented: before sending data over the network, a connection between a sender and a receiver must be established. It stays alive even if there are no data to send and it expires when no more necessary. In this way TCP can create, maintain and close a connection;
- Reliable protocol: the delivery of segments to a destination is guaranteed thanks to the mechanism of acknowledgements, sent from the receiver to the sender;
- Bidirectional service: it allows two applications to send data in both directions at the same time, so that its service can be considered 'Full Duplex';
- Data that have reached the final destination must be in order and sent at most once;
- Flow Control: TCP can control the flow between the sender and the receiver and the congestion over their connection, thanks to the mechanism of the sliding window. This can improve the performances of the buffers of the two end-points.

2.3 TCP segment structure

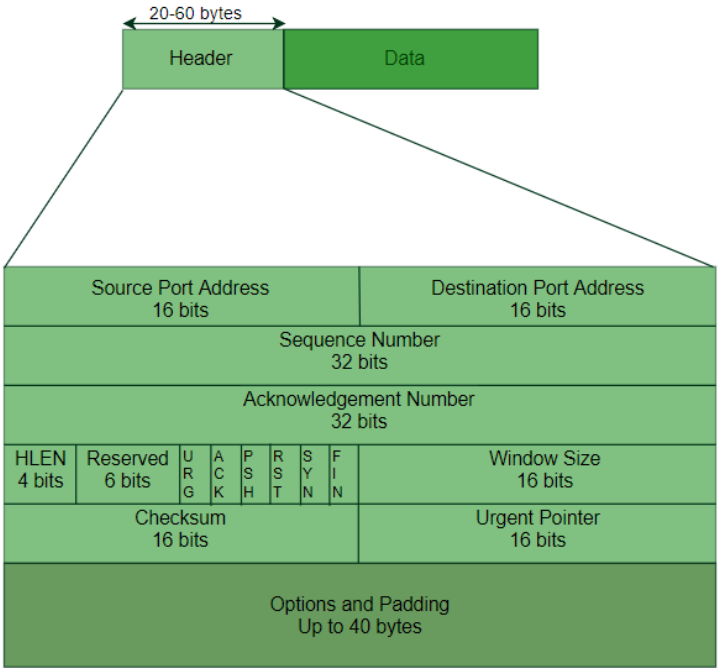

Figure 2.1. TCP segment header (source: GeeksforGeeks)

The TCP Protocol Data Unit (PDU) is called segment and consists of an header and a payload. The segment header contains ten mandatory fields and data stored in it constitute a communication channel between a sender and a receiver. The main of them are:

- Source port (16bit): Identifies the sending port;
- Destination port (16bit): Identifies the receiving port;
- Sequence number (32bit): Has a double role, depending on the value of the SYN flag. If it is set to 1, then this is the initial sequence number and the sequence number of the actual first data byte and the acknowledged number in the corresponding Ack are equal to this sequence number plus 1. If the SYN flag is 0, then this is the accumulated sequence number of the first data byte of this segment for the current session;
- Acknowledgment number (32bit): If the ACK flag is set then the value of this field is the next sequence number that the sender of the Ack is expecting;
- Flags (also knows as Control bits); they are nine but the two most important of them are:

1. SYN: Synchronize sequence numbers. Only the first packet sent from each end-point should have this flag set;
2. ACK: indicates that the Acknowledgment field is significant. All packets after the initial SYN one sent by the client should have this flag set.

- Window size (16bit): The size of the RcWnd, which specifies the number of bytes that the sender of this segment can currently receive.

2.4 Connection establishment

To send data over the communication channel, a connection between the two involved hosts must be established. Before a client attempts to connect to a server, this one must first bind to and listen at a port to open it up for connections. This procedure is called "passive open" and once it is established, a client can initiate an active open. After that, a procedure called *three-way handshake* occurs. It has the aim to establish a reliable connection between the client and server and consists of three different messages:

1. SYN: The client sends a SYN to the server and it sets the segment's sequence number to a random value x;
2. SYN-ACK: In response, the server replies with this message. The acknowledgment number is set to one more than the received sequence number (i.e. x+1), and the sequence number chosen by the server is another random number (i.e. y);
3. ACK: Finally, the client sends an ACK back to the server; the sequence number is set to the received acknowledgement value (i.e. x+1), and the acknowledgement number is set to one more than the received sequence number (i.e. y+1).

At this point, both client and server have received an acknowledgment of the connection. The first two steps establish the connection parameter (sequence number) for one direction. Instead, steps 2 and 3 establish it (sequence number) for the other direction. In this way, a full-duplex communication is set up.
Figure 2.2 shows the previous explanation.

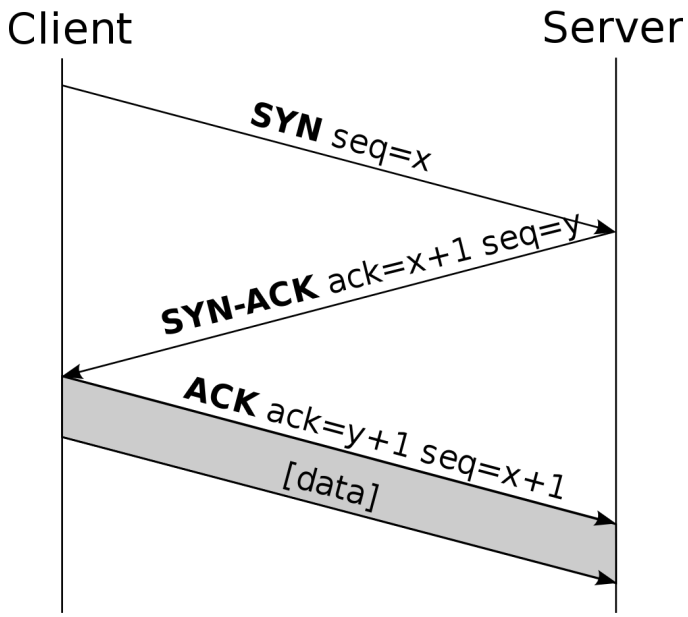

Figure 2.2. Three-way handshake (source: Wikipedia-TCP)

2.5 Data transfer

Regarding data transfer, TCP provides several key features, such as:

- Ordered data transfer: The sequence number is used by TCP in order to identify the data of each byte. By identifying the order of the bytes sent from each host, data can be reconstructed in order, even in presence of any packet loss [3]. Acknowledgements (ACKs) are sent with a sequence number by the receiver of data to tell the sender that data has been received to the specified byte.
- Retransmission of lost packets: Reliability is another particular feature of TCP and it is achieved by the sender detecting lost data and retransmitting it in two different ways: retransmission timeout (RTO) and duplicate cumulative acknowledgements (DupAcks).
    1. Dupack-based retransmission: if a segment in a stream is lost, then the receiver cannot acknowledge packets after it because it uses cumulative ACKs. Hence, the receiver acknowledges the last packet received again on the receipt of another data packet. This mechanism is used as a signal for packet loss and if the sender receives three duplicate acknowledgements, it retransmits the last unacknowledged packet [3];
    2. Timeout-based retransmission: each time the sender transmits a segment, it initializes a timer with an estimate of the arrival time of the acknowledgement. If this timer expires before the sender

receives the ACK, the segment is retransmitted with a new timeout threshold.

The two main aspects of TCP which need a particular attention and a deep study, are the following:

1. Flow Control
2. Congestion Control

2.5.1 Flow Control

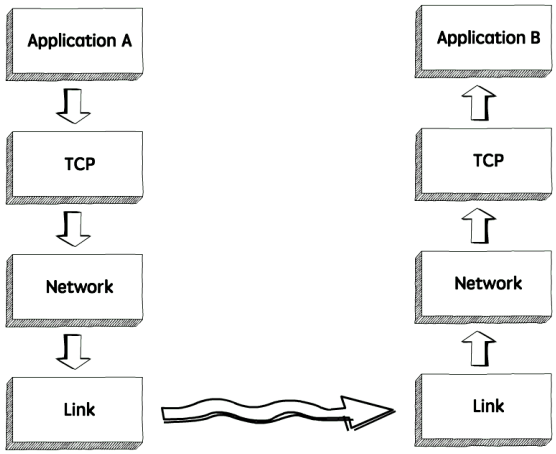

Figure 2.3. Data transmission over TCP (source: Brian Storti's blog)

As shown in figure 2.3 above, when there is need to send data over a network, the sender application writes data to a socket, the transport layer (in our case, TCP) will put this data in a segment and deliver it to the network layer. On the receiver side, the network layer will deliver this piece of data to TCP, which will make it available to the receiver application [2]. Moreover, TCP stores the data to send in the send buffer, and the data it receives in the receive one and when the application is ready, data are read from the receive buffer. Flow Control is about making sure the sender does not send packets when the receive buffer is already full, as the receiver would not be able to handle them and consequently it would drop them. To control the amount of data that TCP can send, the receiver will advertise its Receive Window (Rcwnd), that is, the spare room in the receive buffer[2].

Every time the receiver sends an ack message back to the sender, acknowledging a packet received correctly, the ack message contains also the value of the current RcWnd, so the sender knows if it can keep sending data. RcWnd is a dynamic variable defined as follows:

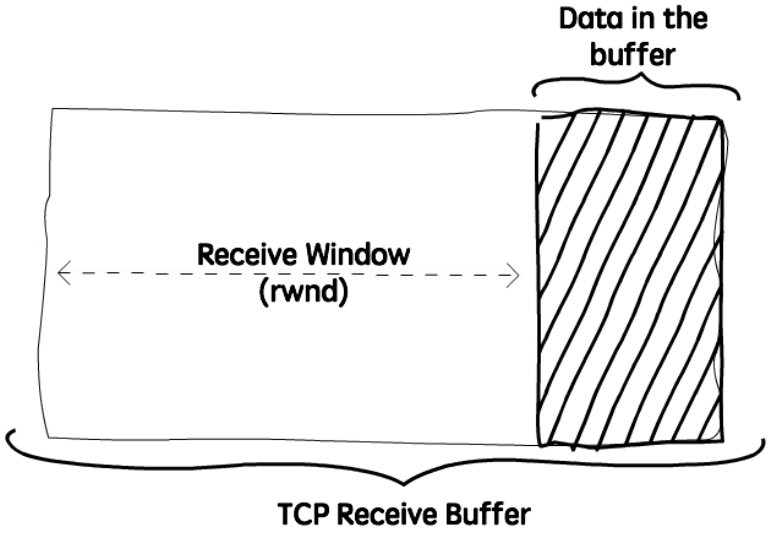

Figure 2.4. RcWnd in the TCP Receive Buffer Brian Storti's blog)

- LastByteRead: number of the last byte in the data flow that the application read from the receive buffer;
- LastByteRcvd: number of the last byte of the data flow coming from the network, copied in the receive buffer.

As TCP aims to avoid the overload of the receive buffer, the RcWnd is defined by the following formula:

$$RcWnd = RcvBuffer - [LastByteRcvd - LastByteRead]$$

where $RcvBuffer \geq [LastByteRcvd - LastByteRead]$ to deny overflow.

2.5.2 Congestion Control

Congestion control is another control provided by TCP in order to guarantee a reliable connection and prevents a node from overwhelming the network. This control of congestion is carried out by acting on the transmission at the end-points thanks to the information deducible from the terminals themselves on the state of the packet transmission. For example, acknowledgments for data sent, or lack of acknowledgments, are used by senders to infer network conditions between the TCP sender and receiver. The Congestion Control mechanism detects a situation of congestion in the terminal entities of the connections and forces each sender a limit on the maximum number of segments to send to the receiver, according to the perceived network congestion. This limit is represented by the Congestion Window (CWnd) and it is maintained by the sender, whereas the sliding window size is maintained by the receiver. In TCP, the Congestion Control mechanism works together with

the Flow Control, which sets an upper limit on the amount of data transmitted and not yet acked by the sender through the RcWnd. Therefore, the amount of data not acked by the sender cannot exceed the minimum between the values of CWnd and RcWnd, that is:

$$LastByteSent - LastByteAcked \leq min(CWnd, RcWnd)$$

The algorithm

Modern implementations of TCP contain four phases for the Congestion Control algorithm:

- Slow start: First, the 'ssthresh' (slow start threshold) variable has to be introduced. The *slow start* phase occurs when the value of CWnd is less than the value of this variable. In the beginning of the transmission the variable is set to a very high value, whereas the size of the CWnd is equal to the size of a segment. During this phase, the CWnd is equal to 1 by default and the TCP sender starts transmitting the first data segment, waiting for an ack. The CWnd increases by an amount equal to the MSS (Maximum Segment Size) for each acked segment. As a direct consequence, for each RTT the CWnd doubles in size. The slow start phase is maintained until a congestion event does not incur, or when the size of the congestion window is greater or equal to 'ssthresh', or in case three equal acknowledges are acked by the sender. In this last case the TCP sender puts its CWnd at 1 and the slow start process starts again. The value of 'sstresh' is now set to CWnd/2. Subsequently we pass to the AIMD phase (when $CWnd \geq ssthresh$) or Fast Recovery (when three acks are identified as duplicates).

- Additive increase/multiplicative decrease (AIMD): This is a control algorithm which combines the linear growth of the CWnd with an exponential reduction when a congestion occurs.

- Fast retransmit and Fast recovery: Phase which reduces the time a sender waits before retransmitting a lost segment. A TCP sender uses a timer to recognize lost segments and if an acknowledgement is not received within that timer, the sender will assume the segment was lost in the network, and will retransmit it. So, the duplicate acknowledgement mechanism is the basis for the fast retransmit phase. When a sender receives three duplicate acknowledgements, it can be sure that the segment carrying the data which followed the last in-order byte specified in the acknowledgment was lost. A sender with fast retransmit phase will then resend this packet immediately without waiting for its timeout. On receipt of the re-transmitted segment, the receiver can acknowledge the last in order byte of data received.

2.6 Congestion Control algorithms

There are several Congestion Control algorithms, each of them present some different peculiarity but during my stage at Clevernet, I had the possibility to study two of them, which are widely used today. These are:

1. TCP Cubic
2. BBR

2.6.1 TCP Cubic

TCP Cubic is the best loss-based Congestion Control algorithm and the idea behind it is that of exploiting today's communications links having higher bandwidth levels. In a network composed by wide bandwidth links, if a Congestion Control algorithm that slowly increases the transmission rate is used, there would be a waste of the capacity of the links themselves.
For this reason the intention of this algorithm is that of having congestion windows with more aggressive incremental processes, but also avoiding from overloading the network. To achieve this, and in order to increase and decrease the transmission ratio, the algorithm uses a cubic function, as shown in figure 2.5 below.

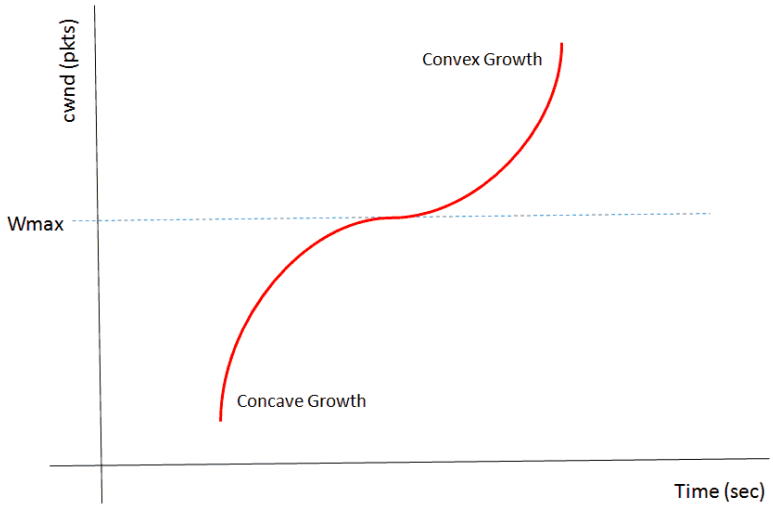

Figure 2.5. The Window Growth Function of Cubic (source: Pandorafms)

The main feature of Cubic is that its window growth function is defined in real-time so that its growth will be independent from RTT and the procedure followed by the algorithm is soon described:

1. When a congestion event occurs, the window size for that instant will be recorded as Wmax(maximum window size);

2. This value is set as the inflection point of the cubic function that will govern the growth of the CWnd;

3. The transmission is then restarted with a smaller window value and, if no congestion has occurred, this value will increase according to the concave portion of the cubic function.

4. As the window approaches Wmax, the increments will slow down;

5. Once the tipping point has been reached (that is Wmax), the value of the window will continue to increase discreetly.

6. Finally, if no congestion occurs, the window size will continue to increase according to the convex portion of the function [8].

Therefore Cubic implements schemes of large increments at first, which decrease around the window size that causes the last congestion, and then continue to increase with large increments. However it acts too aggressively, contributing to self-induced congestion, as intermediate routers start dropping packets causing consistent throughput reduction. But also buffering increases, reducing the responsiveness of latency-sensitive concurrent flows, such as those of interactive applications.

2.6.2 BBR

BBR ("Bottleneck Bandwidth and Round-trip propagation time") is a congestion control algorithm developed at Google. As widely written, Congestion Control algorithms base their work only on indications of lost packets as the signal to slow down. This algorithm differs from the previous one because it is not based on packet loss but it considers how fast the network is sending data. For a given network connection, it uses recent measurements of the network's delivery rate and RTT to build an explicit model that includes both the maximum recent bandwidth available to that connection, and its minimum recent round-trip delay.[4]. BBR then uses this model to control both how fast it sends data and the maximum amount of data to be sent over the network at any time. The advantages of this new model translate into:

- Higher throughput: BBR enables throughput improvements on high-speed links. As it is resilient to losses, a BBR connection can use a path with packet loss, making traffic on backbones faster, and bandwidth significantly increased;

- Lower latency: BBR enables also reductions in latency, especially in those networks connecting users to the internet.

Chapter 3

Tools employed

To pursue the main objective of my Thesis, several tools were useful and interesting, leading us to overcome difficulties and to find efficient solutions. For this reason this chapter is all focused on the tools used during the stage at Clevernet.
The idea is to provide the reader a detailed background about tools and software which helped me to advance in the development of this ambicious project.

3.1 tcpdump

The first tool to describe is tcpdump. It is a packet analyzer that runs under the command line allowing the user to see network packets transmitted or received over a network to which the NIC of the computer is attached. tcpdump basically prints on the standard output or on a .pcap file the contents of network packets, so both ip source and destination, ports on which they are sent, and the layer-5 network protocol. Moreover it can read packets from a network interface card or from a previously created saved packet .pcap file. Here are shown some examples of packets capture:

sudo tcpdump -i eth0 src 192.186.1.185

With this command, the filter will capture all the traffic going from the source (192.186.1.185), which network interface is eth0, to any destination.

sudo tcpdump -i eth0 -w capture.pcap src 192.186.1.185 and tcp

With this command, the filter will capture only TCP traffic going from the source (192.186.1.185), which network interface is eth0, to any destination. The captured traffic is written down on 'capture.pcap' file.

3.2 eBPF/XDP

The second tool is eBPF, which is part of Linux kernel allowing to write programs that run on events like disk I/O, which running in a safe virtual machine in the kernel. Extended BPF derives from BPF, a technology which optimizes packet filters. Trying to filter traffic with tcpdump with an expression (matching on a host or port), it gets compiled into BPF bytecode which is executed by an in-kernel virtual machine.
An eBPF program is attached to a designated kernel event (kprobe) and when it occurs, any eBPF programs attached to it is executed. eBPF is particularly suited to writing network programs and it is possible to write programs that attach to a network socket to filter traffic and to classify it. The XDP (eXpress Data Path) project, in particular, exploits eBPF to do high-performance packet processing by running eBPF programs at the lowest level of the network stack, immediately after a packet is received, therefore before any memory allocation needed by the network stack itself, because memory allocation can be an expensive operation. Every eBPF program must pass a preliminary verifying test before the user loads it in order to avoid executing malicious code in kernel space.
The preverifier checks that the program does not contain out-of-bounds accesses and loops. The main data structure used in all eBPF programs is the map, which is a data structure allowing data, stored and retrieved using a key, to pass between the kernel and user space in a way that they can communicate with each other.
A map is defined by four values: a type, a maximum number of elements, a size defined in bytes, and a key size in bytes. There are different map types and each provides a different behavior and set of tradeoffs.
Moreover, eBPF can be used for software defined networks, DDoS mitigation, intrusion detection and more.

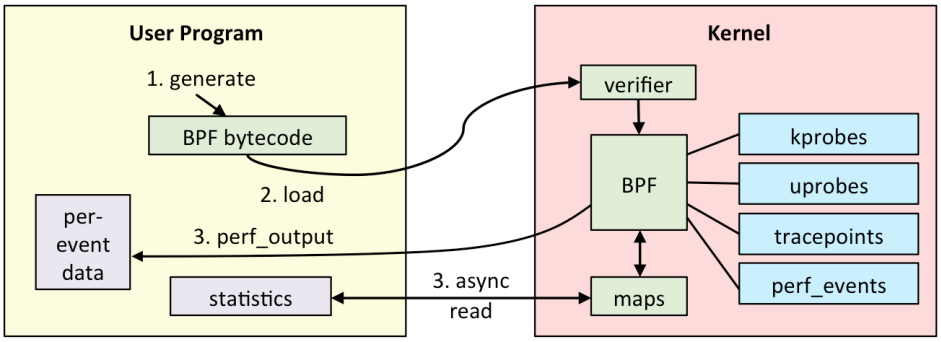

Figure 3.1. eBPF workflow (source: Blog of Brendan D. Gregg)

3.3 libpcap

pcap is an application programming interface (API) for capturing network traffic. In particular, libpcap provides the packet-capture of many open-source and commercial network tools, such as protocol analyzers, network monitors and traffic-generators.
It also supports saving captured packets to a file, and reading files containing saved packets; applications written, using this tool, are useful to capture network traffic and analyze it, or to read a saved capture and analyze it. A capture file saved in the format used by libpcap can be read then by applications which understand that format, such as tcpdump or Wireshark.

3.4 NetEm

NetEm is a tool which enhances the Linux traffic control facilities in a way that the user can add delay, packet losses and more other characteristics to outgoing packets from a previously selected NIC. It provides users several options, but the most used by me were:

- delay: the 'delay' option adds a chosen delay (in ms) to the outgoing packets to chosen network interface;
- rate: allows the user to send outgoing packets at a rate specified in common units, such as kbit or mbit.

Here is an example about how to use this command:

```
sudo tc qdisc add dev eth0 root netem rate 500mbit delay 20ms
```

It allows to delay by 20ms all outgoing packets on device eth0 with a rate of 500 Mbit.

3.5 Token Bucket Filter

The Token Bucket Filter (TBF) is a shaper for traffic control and it guarantees that a configured rate is not exceeded.

3.5.1 The algortithm

Traffic is filtered based on the expenditure of tokens which correspond to bytes, but each packet consumes some tokens. On creation, the TBF is fed with tokens which correspond to the amount of traffic that can be burst and

they arrive at a steady rate, until the bucket is full. If no tokens are available, packets are queued, up to a configured limit.
At this moment TBF computes the token deficit, and throttles until the first packet in the queue can be sent. If it is not possible to send packets at maximum speed, the user can configure a peakrate to limit the speed at which the bucket empties. This variable is implemented as a second TBF with a very small bucket, so that it doesn't burst. The parameters which I used to configure the shaper during my tests were:

- burst: it represents the size of the bucket, in bytes. This is the maximum amount of bytes that tokens can be available for instantaneously;
- rate: which defines the speed which sending outgoing packets;
- latency: which specifies the maximum amount of time a packet can sit in the TBF.

An use-case:
sudo tc qdisc add dev eth0 root tbf rate 500mbit burst 3200b latency 100ms

3.6 D-IGT

D-ITG (Distributed Internet Traffic Generator) is a tool capable to produce both IPv4 and IPv6 traffic. In addition to this, D-ITG can measure several performance metrics (e.g. throughput, delay, jitter, packet loss) at packet level.
D-ITG can generate traffic by using stochastic models for packet size (PS) and inter departure time (IDT) that simulate application-level protocol behavior. At the transport layer, it supports TCP (Transmission Control Protocol), UDP (User Datagram Protocol), ICMP (Internet Control Message Protocol) and more.

3.6.1 The architecture

The main features of D-ITG are provided by ITGSend and ITGRecv. The former is the component responsible for generating traffic toward the latter. ITGSend can send multiple parallel traffic flows to multiple ITGRecv instances, and this last one can receive multiple parallel traffic flows from them and a signaling channel is created between each couple of ITGSend and ITGRecv components with the aim to control the generation of all the traffic flows.
ITGSend and ITGRecv can optionally generate log files containing all information about all packets exchanged. These logs can be saved locally or sent,

through the network, to another component called ITGLog.
Finally, thanks to the ITGDec component, which is in charge of analyzing the log files, the user can read and analyze performance metrics related to the traffic flows.
The user can control each experiment from a single vantage point: the components in charge of receive traffic behave as daemons and are controlled by the other components that want to send traffic to them. In the same way, also the ITGSend components can behave as daemons and controlled through the D-ITG API remotely. A figure which shows the architecture and the functioning of this powerful tool is shown below in 3.2. In particular the different components described above and their way to communicate to each other are shown.

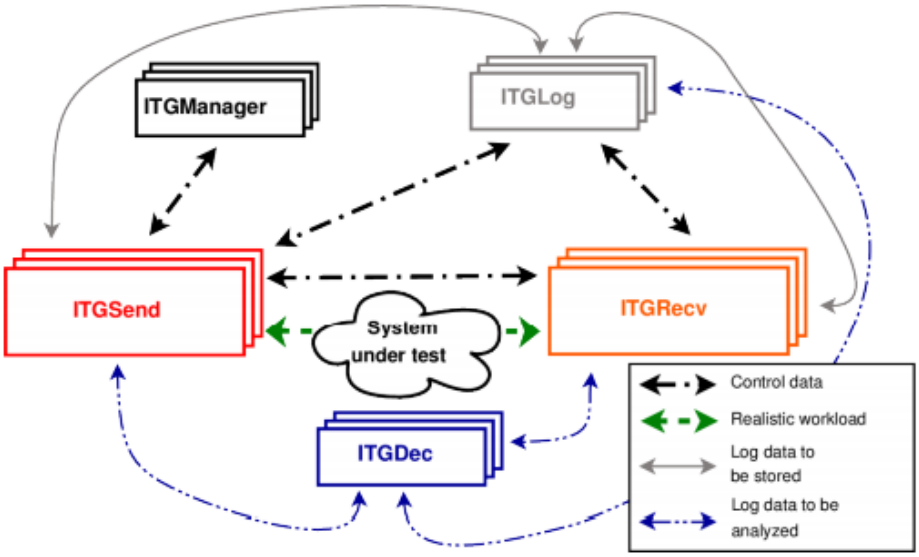

Figure 3.2. D-ITG architecture (source: D-ITG 2.8.1 Manual)

Chapter 4

Network environment and development

4.1 The network infrastructure

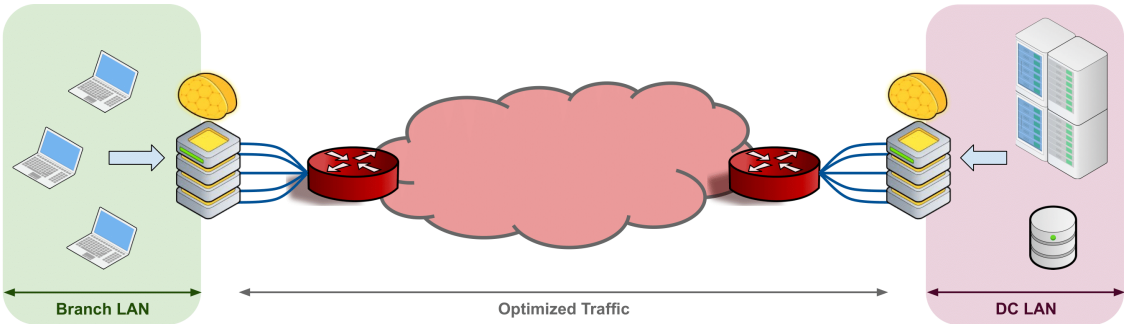

Figure 4.1. A picture of the network topology used

Clevernet has several physical machines where test-beds are installed. A remote ssh tunnel is needed to access such test-beds. During the internship, two of them were used, called, respectively, Arya, whose operating system is Debian, and Deadpool which is powered by VMware, an OS designed exclusively for virtualization. The connection between them is back-to-back because they are in the same DataCenter in Barcelona, therefore they are in the same LAN. On these machines, several test-beds are installed, in particular the *Sigma* test-bed, stored on both machines, which was extensively and exclusively used for this project. Each test-bed is composed by two client-box couples. One couple is installed for each role on a single physical machine. They are Virtual Machines running the Debian distribution of Linux. The client is a VM provided with 2GB of RAM and 1 or 2 CPU cores and it represents a simple workstation laptop. The box is powered by 4GB of RAM and 4 CPU cores and it represents the router. The router aims to always

provide a good, reliable and stable network connection for the client. Therefore, all the tests and research projects involving network deployment are performed on such routers. Moreover edge routers are chosen because they have visibility over all the network flows they are involved into. In order to run tests and network applications, a remote ssh connection to more than one test-bed was necessary, this is why two different machines were required and, as a consequence, two boxes and two clients.

4.2 The initial project

The main project starts from another one already developed which was integrated with some new features and tools. For this reason it was necessary to introduce the "Traffic Generator", which was the starting point of our development. It is a tool entirely coded in Python that allows reproducible traffic to be generated with ease using a configurable and flexible JSON specification file. An orchestrator parses an input JSON that describes an experiment, validates it and runs it. Once done, it will generate a JSON file with the collected data from an experiment. The input JSON contains all the information about an experiment we want to perform, as shown in figure 4.2:

```
{
   "Connection_info":{
      "Client_ID":"sigma-client-deadpool-qa",
      "Client_box_ID":"sigma-box-deadpool-qa",
      "Server_ID":"sigma-client-arya-qa",
      "Server_box_ID":"sigma-box-arya-qa"
   },
   "Generator_type":"Stress_Generator",
   "Iterations":1,
   "Concurrency":2,
   "Path":"1GB",
   "Interval_ms":100,
   "Port":80,
   "Write":false
}
```

Figure 4.2. An example of JSON input file

The most important parameters are the following:

- Iterations: The number of times a number of concurrent transfers are started;
- Concurrency: The number of transfers started for each iteration;
- Interval-ms: The amount of milliseconds between each iteration;

- Port: The port that the HTTP GET request will be sent to;
- Path: Name and relative path within the html directory of the file that has to be downloaded, which is stored on the server host.

All the experiments done concern one or more TCP transfers of a file specified in the 'Path' field, once or more, depending on the value specified in the 'Iterations' field, between two hosts. In the same JSON file described above are specified the roles of the two hosts; this means that one host is the client and the other one is the server. In all experiments the client plays the role of the TCP receiver, whereas the server represents the TCP sender. As shown in the picture above, Sigma client installed on Deadpool is our TCP receiver, whereas the one installed on Arya is our TCP sender. The traffic generated runs across the two boxes, which represent the edge routers for the respective clients. As written in the previous chapter, the connection between Arya and Deadpool is back-to-back as they are in the same LAN, so there are no network delays between them, in particular between Sigma VMs of both physical machines. The traffic generator containing the JSON file and the Python code of the orchestrator are installed on the Arya machine, whereas the code to be executed at the end-points is specified into another tool: the *Traffic Server*. It is coded in Python as well, and it can be installed and executed onto an end-point. It binds to localhost:18891, and an ssh tunnel is required to access it remotely. For this reason two ssh tunnels were opened, one on the Sigma client running on Arya and one on the Sigma client running on Deadpool. With these two connections, the orchestrator can connect to both hosts via other two ssh connections and start its analysis regarding an experiment. At the end of it, a JSON file containing information regarding the test is generated and the ssh tunnels between the orchestrator and the two hosts are closed.

4.2.1 Running an experiment

To run a new experiment, there are several steps to follow, both on the Arya machine and on the two VM clients installed on Arya and Deadpool. First of all two different ssh tunnels to connect to Sigma client on Arya and Sigma client on Deadpool have to be opened. Once in the proper directory which contains the Python code to be executed, the following shell command has to be run on both terminals:

```
sudo python3 server.py
```

where server.py belongs to the *Traffic Server* tool and it is the same Python code that both TCP sender and receiver will execute. After that, the Arya machine's command line, the following command has to be run:

python3 qa-traffic-generator.py [input file.json containing specifics for the test to run]

where qa-traffic-generator.py is the Python code that the orchestrator has to run in order to connect to both end-points via ssh, performing the current experiment and collecting statistics about one or more TCP transfers between the hosts during the execution. The information acquired inside the JSON file generated at the end of an experiment is the following:

- Success: Whether a particular transfer was successful or not;
- Error-String: Only present if a transfer failed. It indicates what failed;
- Time-start: Time at which the transfer started, in order to contextualize it;
- Total-time: Duration of the transfer;
- Average-bandwidth: Bytes transferred per second on average;
- Bandwidth/s: Array of number of bytes transferred each second.

4.2.2 Integrated tools and software

With the tools described, a few initial tests could be performed to better understand how the whole software environment and the interaction between VMs worked. After that, our interest shifted towards collecting advanced statistics, such as information about sent and received TCP packets through a specific port, their timestamps, the Round Trip Time between packet data and their acknowledgements, but also the CWnd and RcWnd read from both the sender and receiver sockets. In order to achieve this, another already developed tool was necessary to be integrated with the ones described above. It is called *tech-ebpf-tools* and it is composed of three main parts:

1. A C file including the eBPF code to be injected in the kernel which recollects stats from data structures and functions return values by means of kernel probes (kprobes);
2. A python script, called *tcpstats.py*, using the BCC library to validate the eBPF code and provide data structures to recollect the stats generated in the kprobes in easy to parse CSV files;
3. Another python script parsing the CSV file and doing data-extraction by means of the pandas library and finally plotting relevant stats;

As it had to be included somehow in the *Traffic Server* tool, two new methods were created in the file *server.py* belonging to it:

- exposed_startEBPFAnalysis (figure 4.3): it takes the input JSON file and the role of the host (sender or receiver) as parameters. Reads the JSON file which includes all the information necessary to identify an host and it stores in local variables the source IP, destination IP and the destination port of the TCP session (if present). Basing on the value of the *role* variable, that tells us if the code is running on the sender machine or the receiver one, it creates different files and stores in a local variable the name of the NIC of the host. Once done, it will run, by the *start* method, the python script which collects stats, but the receiver will run it by managing only one thread which collects information about the TCP session; the sender, instead, will create one additional thread to manage information received by an eBPF/XDP module that I created in order to retrieve timestamp of each packet arriving on the sender NIC. More about this script will be described later;

- exposed_stopEbpfStats (figure 4.4): Once threads have terminated their work, this function is called by the orchestrator. It has the main aim to terminate all threads invoked in the python script. In particular the sender has to stop also the thread which was in charge of running my eBPF/XDP module.

The two functions are shown below:

```
def exposed_startEBPFAnalysis(self, input_json, role):
    myobj = json.loads(input_json)

    if not "source-ip" in myobj:
        src_ip = None
    else: src_ip = myobj["source-ip"]

    if not "dest-ip" in myobj:
        dst_ip = None
    else: dst_ip = myobj["dest-ip"]

    if not "designated-port" in myobj:
        port = None
    else: port = myobj["designated-port"]

    if role == "c":
        output = myobj["output"]
        output += "_client.csv"
        cong_file = myobj["cong_file"]
        cong_file += "_client.csv"
        nic = myobj["rec_nic"]
        tcp_stats = stats.start(role, nic, output, cong_file, interval=0.1, count=-1, src_ip=src_ip, dst_ip=dst_ip, port=port)
    else:
        output = myobj["output"]
        output += "_server.csv"
        cong_file = myobj["cong_file"]
        cong_file += "_server.csv"
        nic = myobj["send_nic"]
        tcp_stats, timestamp, timestamp_event = stats.start(role, nic, output, cong_file, interval=0.1, count=-1,
                                                            src_ip=src_ip, dst_ip=dst_ip, port=port)
        self.timestamp = timestamp
        self.timestamp_event = timestamp_event
    self.tcp_stats = tcp_stats
```

Figure 4.3. startEBPFAnalysis function

```
def exposed_stopEbpfStats(self, role):
    if role == "s":
        self.timestamp_event.set()
        while self.timestamp.isAlive():
            self.timestamp.join(1)

    self.tcp_stats.terminate()
    while self.tcp_stats.isAlive():
        self.tcp_stats.join(1)
```

Figure 4.4. stopEBPFAnalysis function

Having merged these two tools together, several tests could be performed in order to collect statistics at kernel level thanks to the eBPF code, generate CSV files and plot final statistics in an ideal network scenario, that is without cross traffic and 0ms delay between the two hosts.
In figures 4.5 and 4.6 are depicted some graphics obtained thanks to the python script which parses the CSV files generated in *tcpstats.py* and does data-extraction by means of the pandas library. In particular the throughput of a test and the Round Trip Time (RTT) between the two end-points are shown:

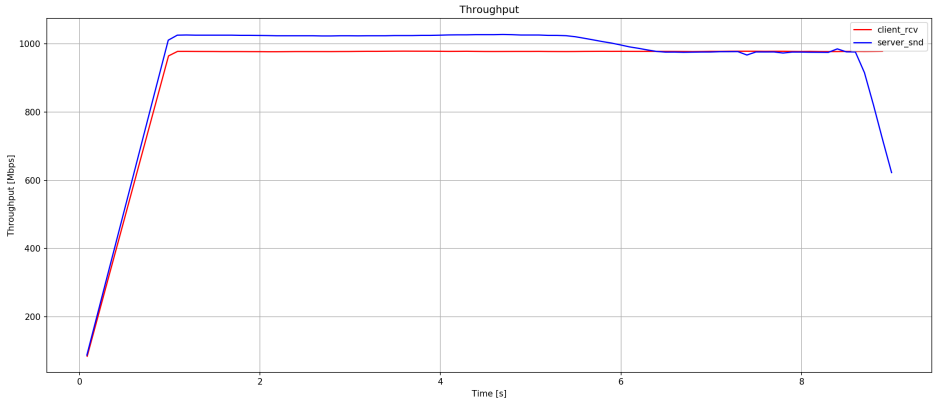

Figure 4.5. TCP throughput

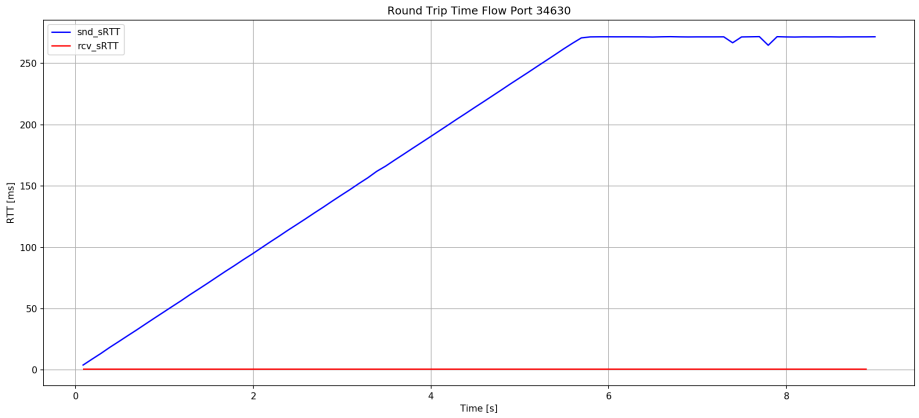

Figure 4.6. Round Trip Time

The throughput was almost 1000Mbps for both sender and receiver, whereas the RTT is always 0 for the receiver side because the pictures refer to an experiment done without delay between the two hosts.

But it was necessary to go further in order to start to approach the project; as a matter of fact there was need to calculate the timestamp for each packet sent and received by the sender in order to later compute an accurate Round Trip Time. This was probably the main part of the Thesis project because a lot of time was spent to find a good way to calculate precise timestamps and, as a consequence, RTT, that was the first factor needed to calculate the Bandwidth Delay Product. For this reason *tech-ebpf-tools* tool had to be integrated with an XDP/eBPF module which is described in the next section.

4.3 A first approach to calculate timestamps

The XDP/eBPF program coded is directly integrated in the *start* function of the *tcpstats.py* tool (figure 4.7). This last one is called by both the sender and the receiver, as shown in figure 4.3 with the aim of acquiring statistics and generating CSV files used by pandas library to plot figures like the previous two shown above, but only the sender will run a second thread, called timestamp, in charge of running the eBPF/XDP module, called *xdp_analysis.py*, to collect information about timestamp of each TCP packet.

The first step necessary to load the C program which had to run the XDP/eBPF code in charge of filtering all TCP packets belonging to an HTTP session between the sender and the receiver. This is done in Python thanks to the *load_func* method, that uses *xdp_myProgram* as the 'main' of the C program using the program tipe BPF. Once done, if this function does not raise

```
def start(role, nic, output, cong_file, interval, count, src_ip=None, dst_ip=None, port=None):
    cond = threading.Event()

    tcp_stats = TCPWndStats(data_file=output, interval=interval, cong_file=cong_file, count=count,
                            port=port, src=src_ip, dest=dst_ip, set_cond=cond)
    tcp_stats.start()

    cond.wait()
    if role == "s":
        event = threading.Event()
        t = threading.Thread(target=xdp.start, args=(event, nic,))
        t.start()
        return tcp_stats, t, event

    return tcp_stats
```

Figure 4.7. start function in *tcpstats.py*

any exception, it is possible to attach the NIC driver to the C program with the *attach_xdp* function. Then, the *open_perf_buffer* function is invoked, which operates on a table defined in BPF as BPFPERFOUTPUT(), and associates the callback Python function *print_event* to be called when the filtered data is available in the perf ring buffer. In this way it was possible to transfer data from kernel to the user space. The fields of each filtered packet we were interested in are shown below in the *TimeLen* structure, which is filled at user space in the *print_event*. These are:

- Both IP source and destination;
- Both TCP source and destination port;
- Acknowledgement number;
- Timestamp of each sent and received packet by the TCP sender

In particular, this last field is retrieved by the *bpf_ktime_get_ns()* helper, which returns a 64-bit unsigned integer. Subsequently, a CSV file is written down with the same header matching information stored in the structure. The code shown below in figure 4.8 helps the reader to have a clearer idea of what described.

```
class TimeLen(ct.Structure):
    _fields_ = [
        ("timestamp", ct.c_uint64),
        ("saddr", ct.c_uint32),
        ("daddr", ct.c_uint32),
        ("sport", ct.c_uint16),
        ("dport", ct.c_uint16),
        ("acknum", ct.c_uint32)
    ]

def run(self, event):
    fn = self.b.load_func("xdp_myProgram", BPF.XDP)

    self.b.attach_xdp(self.nic, fn, 0)
    self.perform_analysis(event)

def perform_analysis(self, event):
    self.b["events"].open_perf_buffer(self.print_event)

    while not event.is_set():
        self.b.perf_buffer_poll()
    self.b.cleanup()

 def print_event(self, cpu, data, size):
    t = TimeLen.from_address(data)
    row = {
        'timestamp': t.timestamp,
        'saddr': ipaddress.ip_address(t.saddr),
        'daddr': ipaddress.ip_address(t.daddr),
        'sport': t.sport,
        'dport': t.dport,
        'acknum': t.acknum}
    self.writer.writerow(row)
```

Figure 4.8. Python code of *xdp_analysis.py* tool

However, Acks accurate timestamping is key, although very complex to achieve in virtualized environments which are very noisy, due to many optimizations in the Linux kernel, such as:

- Interrupt Coalescence: a technique in which events which would normally trigger a hardware interrupt are held back, either until a certain amount of work is pending, or a timeout timer triggers. Used correctly, this technique can reduce interrupt load by up to an order of magnitude, while only incurring relatively small latency penalties. Interrupt coalescing is a common feature of modern NIC [13];

- Generic Receive Offload: GRO is a widely used SW-based offloading technique to reduce per-packet processing overheads. By reassembling small packets into larger ones, GRO enables applications to process fewer

large packets directly, thus reducing the number of packets to be processed [14].

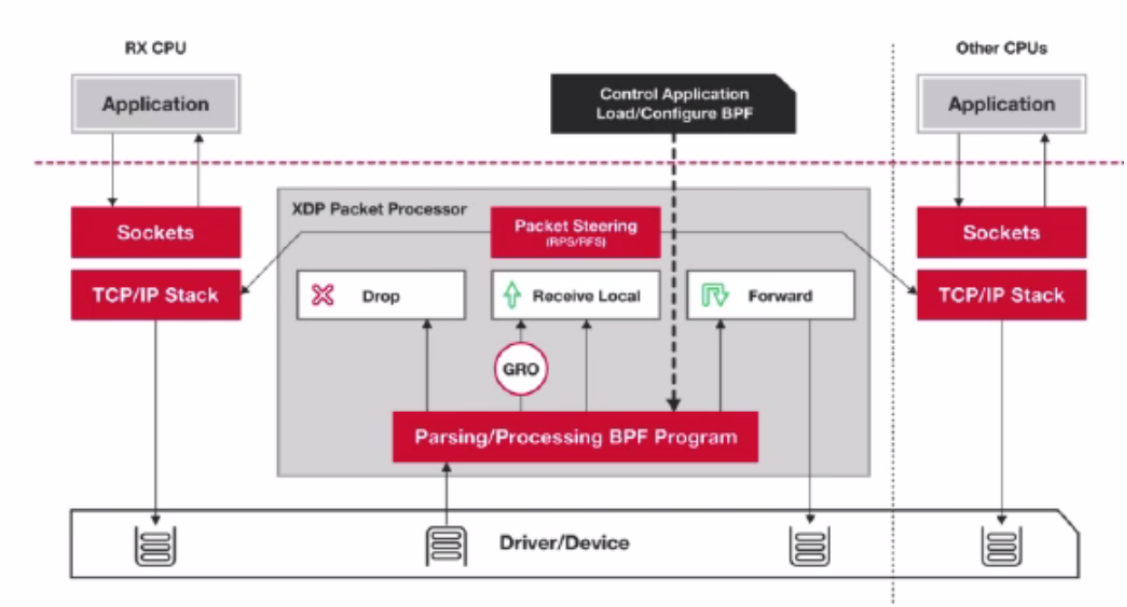

Figure 4.9. XDP processing stack (source: the XDP technology)

These were two key points for our studies; in particular, as shown in the picture above, XDP is put right before GRO and as the code developed worked with XDP, it took all raw packets and we could better understand how the Interrupt Coalescing could have helped us to compute accurate timestamps.
From these statistics it was possible to compute offline, thanks to an open-source Python library called Pandas, the Inter-Packet rate of the packets being acknowledged by their Acks by the following formula:

$$\text{IP rate} = \frac{Ack_i - Ack_{i-1}}{Ts_i - Ts_{i-1}}$$

It is computed by the difference between each ack and its previous (that is the number of Bytes of each packet acked) divided by the difference between the timestamps when the respective packets were acked. In the following a Cumulative Distribution Function for the inter-packet rate is shown, useful to evaluate if our estimations about timestamps were correct or not. It is computed from TCP ACKs timestamp taken in the sender with XDP after a test performed with the following specifications:

- 1 TCP flow with a throughput approximately equal to 1Gbps;
- no Cross traffic;
- no delay between the two end-points

On the x axis of the next figure, the throughput is put, whereas on the y one the probability of having information at certain throughput is set. The result obtained is the following:

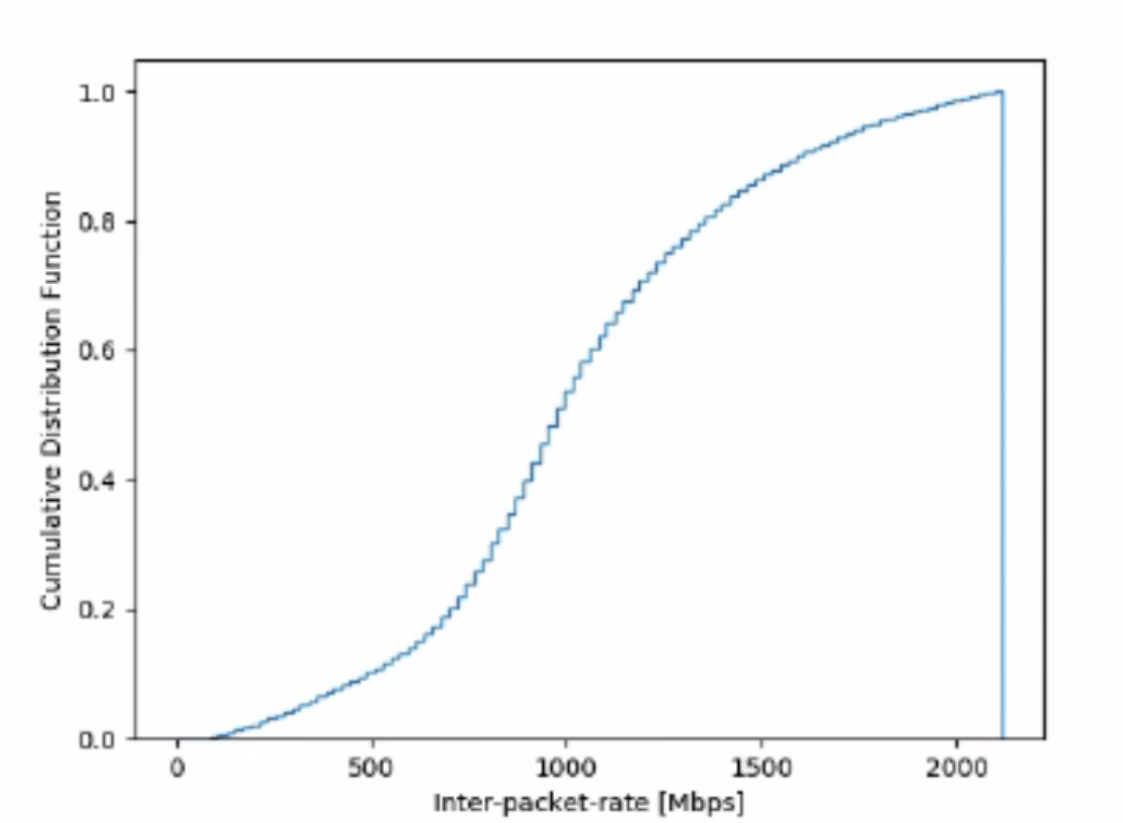

Figure 4.10. Cumulative Distribution Function using XDP

As clearly shown in figure 4.10, the estimations regarding timestamps were not precise because more than 50% of the measurements were higher than the actual maximum capacity: the measurement is invalid. This simply means that the timestamps calculated thanks to the helper were not accurate, because of driver level optimizations. An example of what could have happened is the following: for each n packets an interrupt occurs, XDP catches them all and calculates the timestamp for each single packet in a fast way (around 2Gbps).

4.3.1 Problem of timestamping in XDP

As written above, with XDP we could not get accurate timestamps. The main reason is that each timestamp is associated to the software elaboration of XDP, rather than the exact moment when the packet arrives to the physical NIC of the host. XDP is close to the NIC driver, meaning that timestamps could be accurate, and for this reason an attempt was done using that technology. But as the hardware timestamp is in a struct called

socket buffer (sk_buff) and, unfortunately, XDP does not have access to this structure because sk_buff struct is not allocated yet in the moment XDP is used, another strategy was necessary.

4.4 From XDP to Libpcap

As XDP timestamps were affected by driver level optimizations, we decided to change our strategy to compute them. We actually used Libpcap because we realized that timestamping was very precise with pcap traces. In particular, we digged deeper to find out that Libpcap is actually taking by default, when the hardware and drivers support it, the timestamp taken by the NIC. By running the same code to evaluate the Cumulative Distribution Function, with the same scenario as before, the final result and the measurements were impressive and very precise, as shown below in figure 4.11:

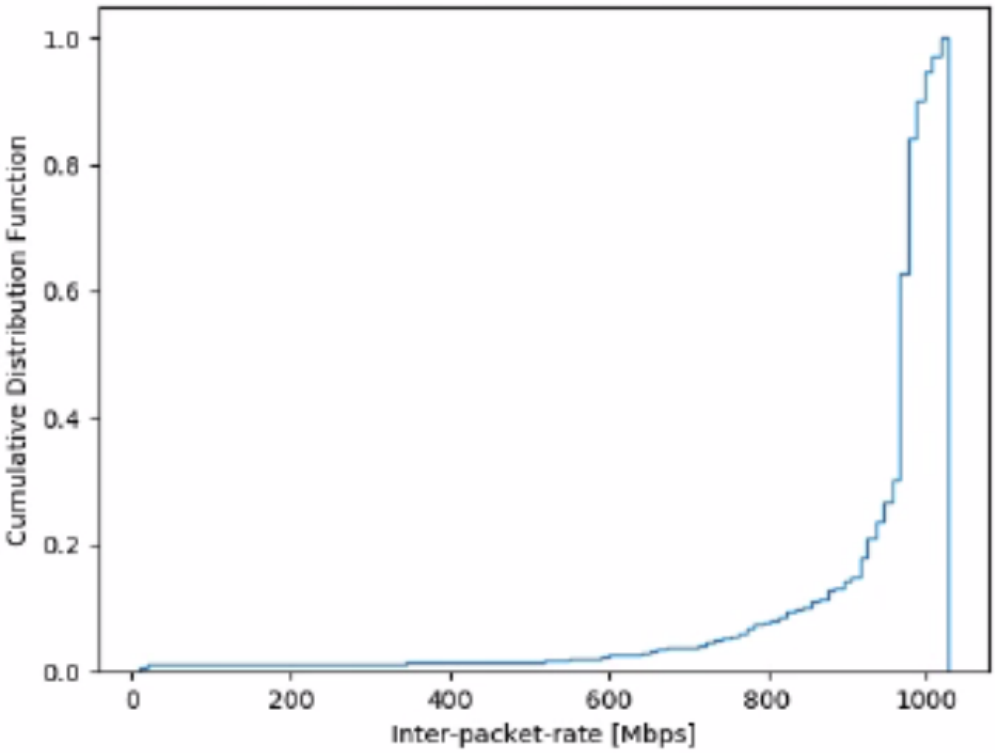

Figure 4.11. Cumulative Distribution Function using Libpcap

The measurements were better and preciser than those obtained with XDP in fact they were so close to the actual maximum capacity (1Gbps). Once noticed that Libpcap worked better than XDP, code in the *Traffic Server* tool had to be changed, in particular the *exposed_startEBPFAnalysis* method

needed some modifications. Before one additional thread to run the XDP/eBPF module was necessary, but this time it was no longer helpful, and for this reason both the sender and the receiver run the same *start* function (figure 4.12) which is included in the *ebpf_tech_tool*. This function is sensibly different from the previous one showed before:

```
def start(role, nic, output, cong_file, interval, count, src_ip=None, dst_ip=None, port=None):
    cond = threading.Event()

    tcp_stats = TCPWndStats(data_file=output, interval=interval, cong_file=cong_file, count=count,
                            port=port, src=src_ip, dest=dst_ip, set_cond=cond)
    tcp_stats.start()

    cond.wait()

    if role == "s":
        pcap_file_received = open("stats_received.csv", "w")
        writer = csv.DictWriter(pcap_file_received, fieldnames=PCAP_HEADER)
        writer.writeheader()
        pcap_file_received.close()
        subprocess.Popen("ebpf_tools/xdp/./parsinganalysis")

    return tcp_stats
```

Figure 4.12. The new start function in *tcpstats.py*

If the code is running by the TCP sender, a CSV file with the following header is created:

- Round Trip Time;
- Timestamp;
- Source and destination IP;
- Source and destination port;
- Sequence number and ack number

Once done, with the python class *subprocess* the Popen method that runs a child program in a new process is invoked. The program is the a one that I have developed entirely in C and captures packets thanks to the Libpcap library.
As shown in figure 4.13, it works in the following way: *pcap_lookupdev* function included in Libpcap library is used to read the physical NIC of the sender, subsequently *pcap_open_live* function was necessary to obtain a packet capture handle to look at packets on the network. The variable *dev* specified in it represents the NIC read by the previous function and is used to specify the network device to open. Finally *pcap_loop* processes packets from a live capture and the most important parameter passed to it is without any doubt the *pkt_handler* which is the callback to the function to be invoked each time a packet is captured on the NIC.

```
int main()
{
    pcap_t *fp;
    char *dev, errbuf[PCAP_ERRBUF_SIZE];

    dev = pcap_lookupdev(errbuf);
        if (dev == NULL)
        {
                fprintf(stderr, "Couldn't find default device: %s\n", errbuf);
                exit(1);
        }

    fp = pcap_open_live(dev, 100, 0, -1, errbuf);
    if (!fp)
    {
        fprintf(stderr, "Couldn't open device %s\n", dev);
            exit(1);
    }

    pcap_loop(fp, -1, pkt_handler, NULL);
    pcap_close(fp);

    return 0;
}
```

Figure 4.13. main function of the C program

Each TCP packet belonging to the HTTP session between the sender and the receiver is taken; in particular a filter is set on the source and destination port in order to distinguish if the packet contains data or an acknowledge. Once acquired that information, in a structure are stored the main fields of the packet, which are:

- Both source and destination IP addresses;
- Both source and destination port;
- Packet length;
- Sequence number and Ack number;
- Timestamp related to each packet;

This last ones have been calculated by the struct *timeval* included in the struct *pcap_pkthdr*, with a precision of microseconds. Each time a packet is received by the sender, RTT by matching the last ack number received with one of the previous sequence numbers of packets sent to the receiver could be computed. When $Ack\_number = Seq\_number$ the RTT could be calculated by subtracting the timestamp of the ack number to that belonging to the sequence number. RTT of each acked packet was also stored in the same structure mentioned above and for each thousand of packets acked, a thread in charge of writing on a CSV file the whole information saved in the structure is invoked. The file generated was then useful to compute, by another tool using the python pandas library, an estimation of the Available Bandwidth offline.
Each test performed was combined together with different .pcap traces obtained by running "tcpdump" on strategic hosts in the network because it

was important to check if the estimation of the Available Bandwidth generated after acquired information from the CSV file was enough similar to that collected on the .pcap traces. Although the code developed was working properly for our purposes, we wanted it to be more general, especially regarding the filter set. As a matter of fact, the previous filter based only on the source and destination port could not work for all the applications; for this reason we needed another way to distinguish sending and receiving flow in a more dynamic way.
The first thing to do was finding dynamically the direction of the two flows and for this reason three different data structures were created. One is shared between the two flows and it is filled until both directions were not found. To do so, the only filter set regarded the layer-4 protocol; if it is equal to TCP, the shared struct is filled with around 30 packets and then it is checked the sequence numbers for the same couple of src and dest IP and source port and destination port. If more than three equal sequence numbers were found, it could be inferred that the couple of IP addresses and ports belonged to the receiving flow, as a consequence the sending one is found by inverting the IP addresses and ports. Once acquired the information about flows, only the other two structures were filled with the information regarding, respectively, a packet data or an acknowledgement. The information collected in the structures are basically the same described above, and the CSV file is written down by a secondary thread for each thousand of packets acked.
However we found out that the program worked slow because of the big amount of data structures used and the total complexity of the final algorithm. For this reason C++ was chosen, as it allows the programmer to use collections, such as maps, rather than struct in a smarter and faster way. The idea about how to develop the algorithm was the same as described before but instead of using three struct, three map were used, one shared to be filled until the flows were not found and two used to be filled with, respectively, information about data packets and acknowledgements. Maps are collections made up of two parts: key, which is always unique, and value. In this case the key was formed by the four-tuple (source IP, dest IP, source port, destination port) and the value was a vector of concatenated strings. In it the following fields are put:

- timestamp of each packet arriving on the NIC;
- sequence number;
- ack number

With the same strategy, the direction of the two flows was found and the vector of the other two maps was filled, rather than filling the shared one. To have a concrete idea on the algorithm developed to find both flows, figure 4.14 is shown below:

```
for (auto &v : values) {
    found_receiver = 0;
    equal_seq_num = 0;

    std::vector<std::string> splitted = split(v, ',');
    std::string seq_num = splitted.at(1);
    for (auto &v1 : values) {
        splitted = split(v1, ',');
        if (splitted.at(1) == seq_num)
            equal_seq_num++;

        if (equal_seq_num > 3) {
            found_receiver = 1;
            break;
        }
    }
    if (found_receiver) {
        sender[inverted_key] = it->second;
        count_sender[inverted_key] = pkts_per_flow[inverted_key];
        compute_rtt(new_key, values, it->second);
        sender[inverted_key].clear();
        break;
    } else {
        sender[new_key] = values;
        count_sender[new_key] = pkts_per_flow[new_key];
        compute_rtt(inverted_key, it->second, values);
        sender[new_key].clear();
        break;
    }
}
```

Figure 4.14. Piece of code to identify data or acks direction

Both flows are immediately differentiated and the other two maps, called, respectively, *sender* and *receiver* are filled. It is also called the *compute_rtt* function which was in charge of computing the Round Trip Time for each acked packet until that moment. Then, the two structure with all the parameters necessary are filled; in particular for each packet received by the sender the *compute_rtt_for_one_value* function is invoked, in charge of computing the Round Trip Time for the last ack number received with the same strategy mentioned above. In a first moment, this function was developed as shown in figure 4.15:

```
void compute_rtt_for_one_value(const std::string& key, long local_ts, std::string ack_num, const std::vector<std::string>& val_sender) {
    std::vector<std::string> splitted;
    std::string value;
    long rtt;

    for(auto it = val_sender.begin(); it != val_sender.end(); it++) {
        splitted = split(*it, ',');
        std::string seq_num = splitted.at(1);

        if (ack_num == seq_num) {
            ts_sender = std::stol(splitted.at(0));
            std::string ack_sender = splitted.at(2);
            rtt = local_ts - ts_sender;

            value = std::to_string(rtt) + "," + std::to_string(local_ts) + "," + ack_sender + "," + ack_num;

            m.lock();
            receiver[key].push_back(value);
            m.unlock();
            val_sender.erase(val_sender.begin(), it);

            break;
        }
    }
}
```

Figure 4.15. Final version of code to compute RTT

For the last ack number received, a for-loop on the whole vector of the sender side is performed in order to find the sequence number that matched the last acknowledgement. When the condition was satisfied the RTT is computed and a new value (composed by the RTT, the timestamp regarding the last ack number received and the ack number itself) is then inserted in the vector of the receiving flow. However this solution was not so efficient in term of complexity because the time spent to iterate over the vector was very higher and we understood that we lost several packets arriving from the network by looking at plots we generated to estimate the Available Bandwidth. For this reason the algorithm to compute RTT needed to act faster and the solution found was that of substituting the vector with an unordered set. In this way it was no longer necessary to iterate over a loop because the complexity to find an element in an unordered set is O(1). This final solution was kept because it worked way better and after around five hundred packets acked, a thread in charge of writing on the CSV file the information about the TCP flow is invoked. With this CSV file we generated plots with Pandas library and we clearly saw that they had more information than before, as shown in the two pictures below.

4.4.1 Comparison between the two methods

On the X axis timestamp in seconds is set whereas on the Y one the IPT acks (in Mbps) is represented. The orange flows refer to the .pcap file captured on the NIC of the TCP sender which starts three parallel TCP transferences with the same receiver. As shown in both figures (4.16, 4.17), a significant amount of IPT acks is around 250 Mbps and this means that the estimation of the Available Bandwidth by means of IPT acks is very precise if we consider that we performed the test by setting the maximum throughput at 300 Mbps without any kind of cross traffic.

The black flows refer to the statistics collected in the CSV file generated from my program with the first approach. In figure 4.16 clearly shows that there is an huge lack of information compared with the one collected from the .pcap file.
Once improved the performance of my algorithm, a test with the same settings as before has been performed. Results from the .pcap and CSV files have almost the same full information about the three TCP transferences, as shown in figure 4.17. This result led us keeping my improved algorithm to compare statistics from both types of file and to perform other tests to estimate the Available Bandwidth.

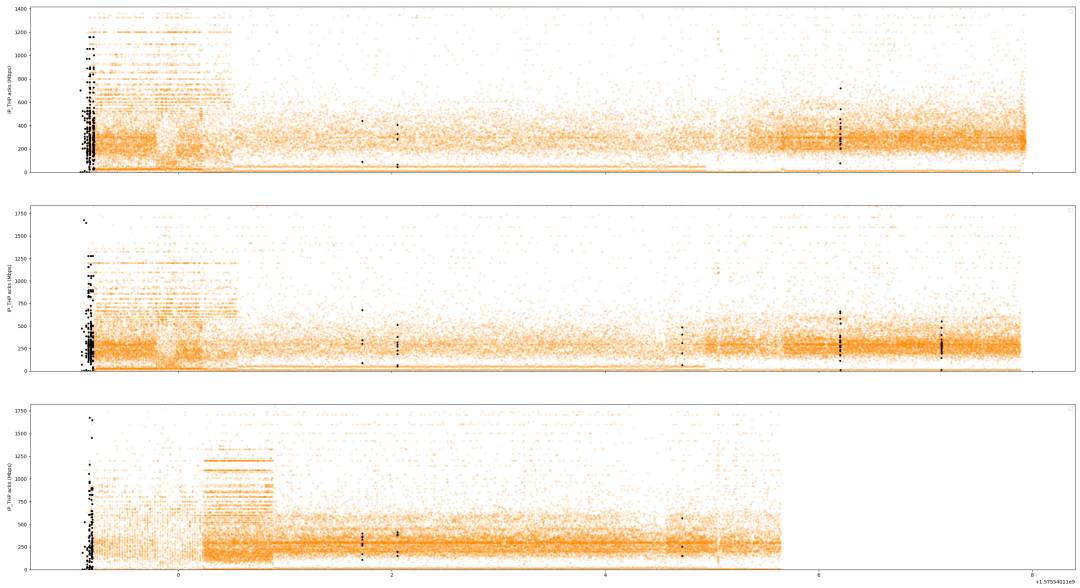

Figure 4.16. Comparison between IPT obtained from .pcap and CSV with the first method

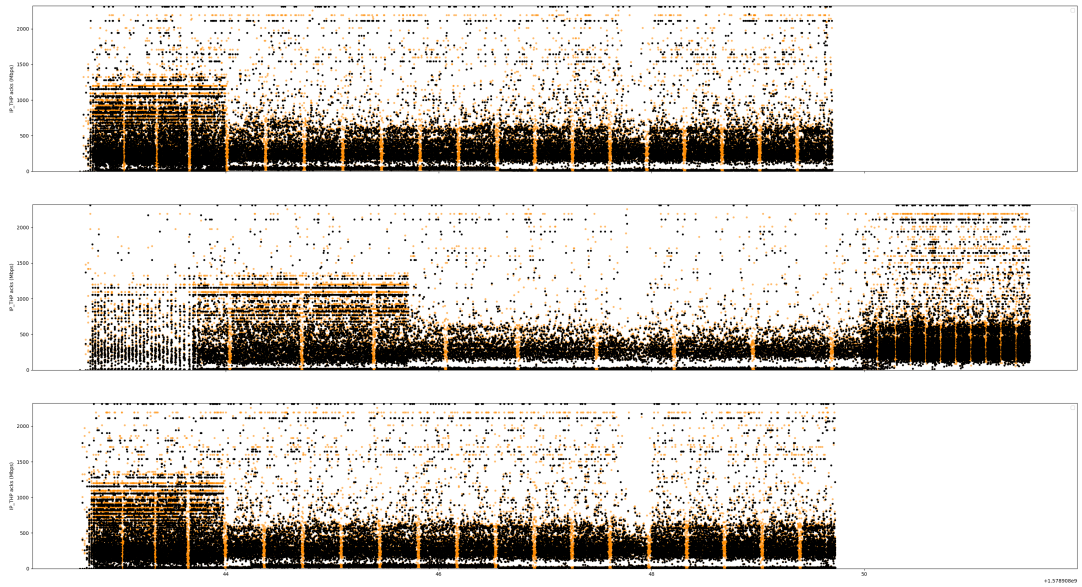

Figure 4.17. Comparison between IPT obtained from .pcap and CSV with the second method

4.5 Tests and measurements

The tests performed were different and various in order to simulate in the best way all the possible network scenarios during one or more TCP transferences. A tuning on several parameters was necessary because it was important to explore and study all different study cases to run the tests. For this reason one of the first parameter which needed to be changed was the delay between the two TCP end-points. Accordingly, tests have been performed with 20ms, 40ms and 120ms delay by tuning it with the netem tool. Due to the fact that a network environment without any kind of cross traffic acting during one or more TCP transferences is not realistic, two new virtual machines were setted up, one in the same LAN of the TCP sender, the other one in the LAN of the TCP receiver. In the picture below is so depicted an example of the complete scenario on which tests were performed. The two VM at the top of the corners represent the TCP sender and receiver, whereas those at the bottom exchange TCP or UDP cross traffic between them.
An example of testbed topology is shown below in figure 4.18

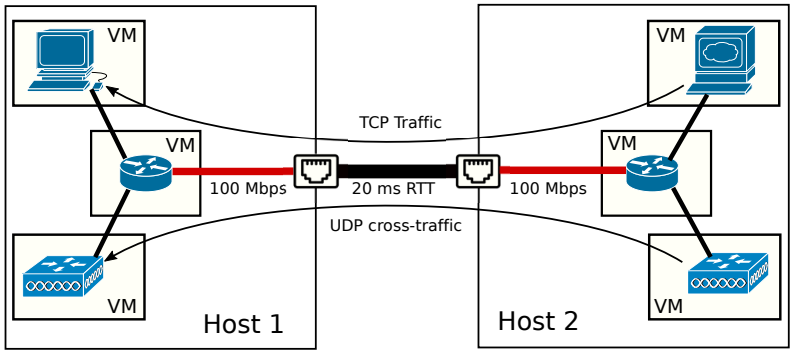

Figure 4.18. Testbed topology with one single 100 Mbps bottleneck

These new machines, which runs, respectively, in Arya and Deadpool hosts, are provided with the D-ITG tool in charging of manually running the cross traffic during a test; sometimes it was so huge in order to make the test very noisy to prove if the tool gave us good estimations of the Available Bandwidth even in extreme conditions. Another parameter tuned was the maximum throughput at which send packets at both senders side and its edge router to better collect statistics and having a clearer plots about the estimations. As a matter of fact the maximum throughput was decreased from 1Gbps to 100Mbps on both senders and on their same edge router, as shown in the picture above, meanwhile the name and relative path from the html directory in the server host of the file that had to be downloaded had a size of 10MB. In the pictures below are shown only two of tests performed. In particular, in figure 4.19 are represented three parallel TCP flows going from the same source to the same destination with one TCP flow of cross traffic going in the same direction. Both the axes are the same described for figures 4.16 and 4.17. As already written above, the plots contain all we have captured from both CSV and .pcap file related to the NIC of the sender. The red line represents the estimation of the Available Bandwidth collected from the .pcap file, whereas the black one is related to the CSV file. As they are quite similar, it can be inferred that the IPT collected from the CSV is correct, this is the Available Bandwidth computed from the IPT of the .pcap is equal to the that one computed from the CSV.
In figure 4.20 are instead shown three parallel TCP transferences in presence of a big amount of UDP cross traffic going in the same direction of the TCP flows.
The red and black lines are, respectively, taken from the .pcap and CSV file and as they are quite equal even in this case, the metric calculated on the IPT acks (represented with blue points) is precise and accurate to estimate the Available Bandwidth.

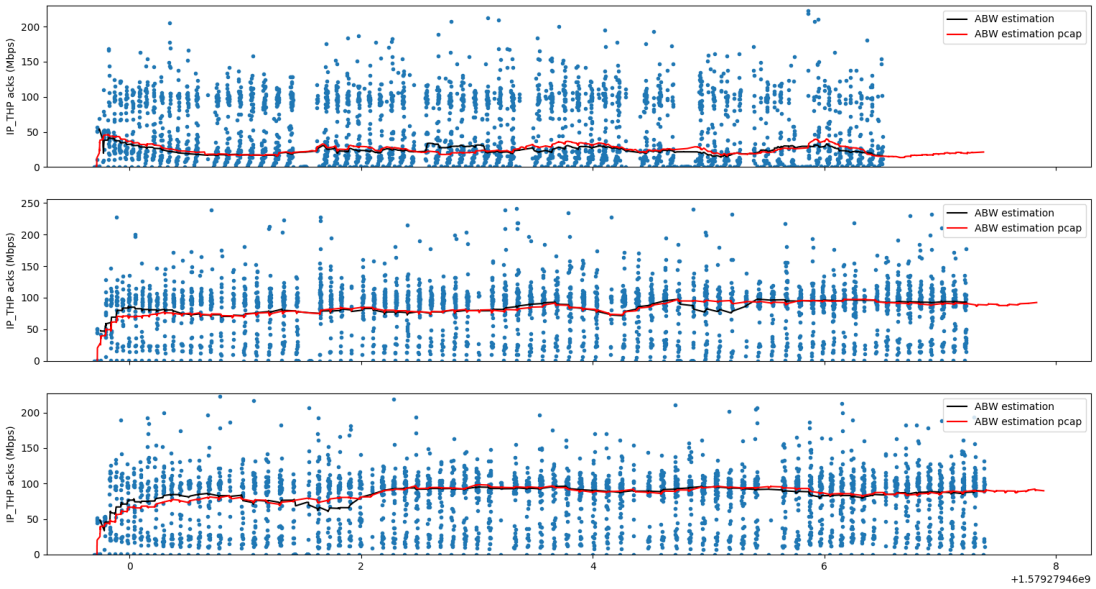

Figure 4.19. Cumulative Distribution Function using Libpcap

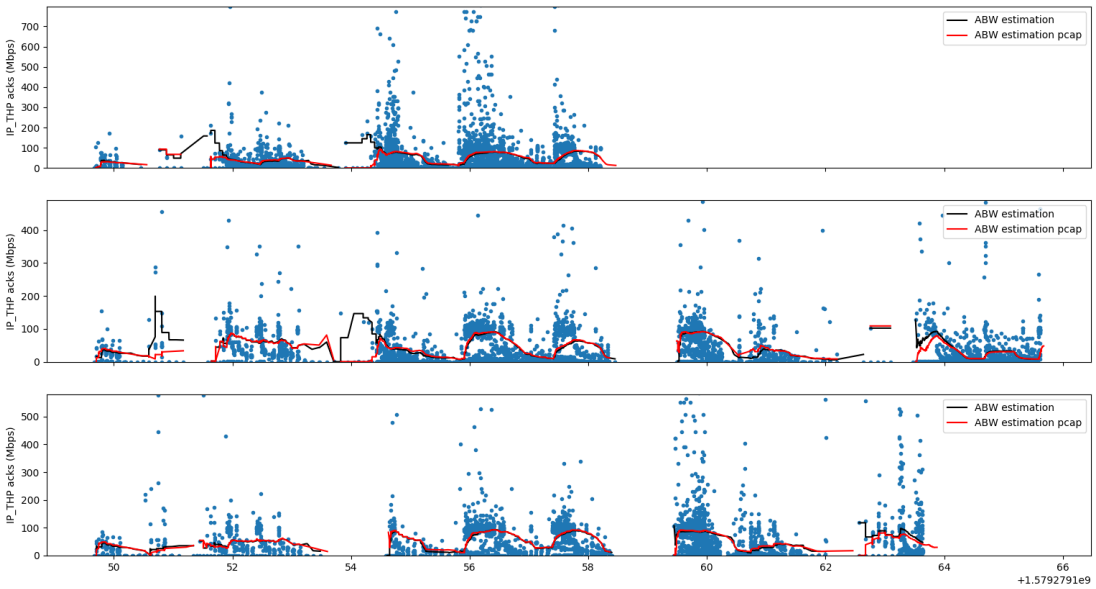

Figure 4.20. Cumulative Distribution Function using Libpcap

Chapter 5

Real-time implementation

Once realized that our offline results were promising with the second version of the C++ code, it was the time to try to implement a new code version which estimated the Available Bandwidth in real-time.
Starting from the previous code, it was only necessary to change the code to be run by the secondary thread, which was previously in charge of writing the CSV file. The largest values of the timestamp and the RTT are taken by the set containing these values plus the ack number of each acked. Elements of the set whose difference between the last timestamp and their current timestamp value is greater or equal to 1 seconds are instead erased.
An iteration in reverse order is then performed on this set and until the condition

$$last\_ts - current\_ts \leq last\_RTT$$

is not satisfied.
Thus, the position of the set not satisfying the condition is saved together with the ack number and timestamp regarding that specific packet.
Another iteration in order over the set is then performed from the saved position until the end. Each value of current timestamp and ack number is taken together with their respective previous values and the following formula could be computed:

$$\text{IP rate} = \frac{Ack_i - Ack_{i-1}}{Ts_i - Ts_{i-1}}$$

The difference between the two timestamps is then divided by 1000000 in order to switch from microseconds to seconds and each computed value of IP rate is pushed into a vector. This is then passed to the *computeQuantiles* function in charge of calculate its quantiles.
They are points of a distribution that relate to the rank order of values in that distribution. In our case, the distribution was the vector containing the IP rate. This sample had to be sorted by definition in order to take the interested percentiles. In particular we were looking for three different of them: the 25th, the 50th and the 75th which could be retrieved by the

respective positions in the vector.
Finally it was possible to calculate the *Interquartile range* (IQR), defined as the difference between the 75th quantile and the 25th one. With it, an upper_limit and a lower one could be defined by the following two formulas:

$$upper\_limit = (1.5 * IQR) + 75th\_quantile$$

$$lower\_limit = 25th\_quantile - (1.5 * IQR)$$

Values of IP rate greater than the upper_limit or less than the lower_limit are then removed from the vector. This function, whose implementation is shown in figure 5.1 returns two values:

- The 50th percentile by value;
- The IQR by reference

```
double computeQuantiles(std::vector<double>& values, double &iqr_to_return) {

    std::sort(values.begin(), values.end());
    int vector_size = values.size();
    ++vector_size;

    float q25 = vector_size*0.25;
    float q50 = vector_size*0.5;
    float q75 = vector_size*0.75;

    int q25_index = ((int)(q25*10))%10;
    q25_index >= 5 ? q25_index = (int)q25+1 : q25_index = (int)q25;

    int q50_index = ((int)(q50*10))%10;
    q50_index >= 5 ? q50_index = (int)q50+1 : q50_index = (int)q50;

    int q75_index = ((int)(q75*10))%10;
    q75_index >= 5 ? q75_index = (int)q75+1 : q75_index = (int)q75;

    --vector_size;
    if (q50_index == vector_size) --q50_index;
    if (q75_index == vector_size) --q75_index;

    double q25_value = values.at(q25_index);
    double q50_value = values.at(q50_index);
    double q75_value = values.at(q75_index);

    double iqr = q75_value - q25_value;
    iqr_to_return = iqr;

    double upper_limit = (1.5*iqr) + q75_value;
    double lower_limit = q25_value - (1.5*iqr);
    values.erase(std::remove_if(values.begin(), values.end(),
    [&upper_limit, &lower_limit](auto e){return e > upper_limit || e < lower_limit;}), values.end());

    return q50_value;
}
```

Figure 5.1. Quantiles

The first time the secondary thread is launched, it takes the largest value of the IP rate from the vector, then it fills an array of IP rate of 10 elements, whose first value is the largest taken before divided by 10, called *val*. Each next value of this new array is given by the sum of the previous element and *val*.
A nested loop of the vector and this new array is performed, and each time the IP rate of the vector is less or equal than any of the elements in the array, the number of occurrences in the same position of the element is increased by one. By iterating over this last array of occurrences and taking its maximum value, the capacity of the link is set to maximum element of the array of IP rate in the same position of the one of occurrences.
By the second time the thread is launched, it is possible to estimate the Available Bandwidth by the computing average on the values of the IP rate belonging to the vector.
This is possible only if two conditions are satisfied:

1. The difference of absolute value between the average of the vector and the 50th percentile is less than 0.1*capacity estimated the first time;

2. the IQR is grater than 0.3*capacity

```
if (found_cap) {
    if (diff_mean_50q < 0.1*capacity && iqr > 0.3*capacity) {
        awb = computeMean(ipt_thp_vector);
        fprintf(f, "%ld %lf\n", last_ts, (awb*8)/1000000);
        fclose(f);
    }
}
else {
    found_cap = true;
    int max = -1;
    for (int i = 0; i < BIN_SIZE; i++) {
        if (occurrences[i] > max) {
            max = occurrences[i];
            capacity = ip_rate[i];
        }
    }
}
```

Figure 5.2. Estimation of link capacity or Available Bandwidth

Figure 5.2 shows the last piece of code described above, in particular *found_cap* variable is a boolean set to false by default and then set to true once the secondary thread has estimated the capacity the first time.
Variable *diff_mean_50q* and *awb* are computed as shown in figure 5.3, respectively by the function *computeDiffMean* and *computeMean*.

```
double computeMean(std::vector<double> &values) {

    auto n = values.size();
    double average = 0.0;

    if (n != 0) {
        average = std::accumulate(values.begin(), values.end(), 0.0) / n;
    }

    return average;
}

double computeDiffMean(std::vector<double> &values, double &q50) {

    double average = computeMean(values);
    return std::abs(average - q50);
}
```

Figure 5.3. Computation of Available Bandwidth and diff_mean_50q

5.0.1 Results

An estimation of the method explained above are shown in figure 5.4 and 5.5. The green line represents the real Available Bandwidth, given by the difference between the capacity estimated on the first one-hundred TCP packets and the cross traffic generated (the yellow line).
The blue one represents the TCP flows on which the estimation is based whereas the red crosses represent the actual estimations of the Available Bandwidth. The majority of them are close to the real one, computed by the difference between the estimated initial capacity and the amount of cross traffic sent on the network. It can be so inferred that our method and measurements are quite precise and accurate. For each estimation, in fact, the error compared with the real Available Bandwidth is computed thanks to the following formula:

$$err = |\, estimatedAwb - realAwb \,|$$

By computing the average of each error we obtain the mean absolute error and, from this, we saw that our results were still promising because this value was around 30Mpbs in the worst case.

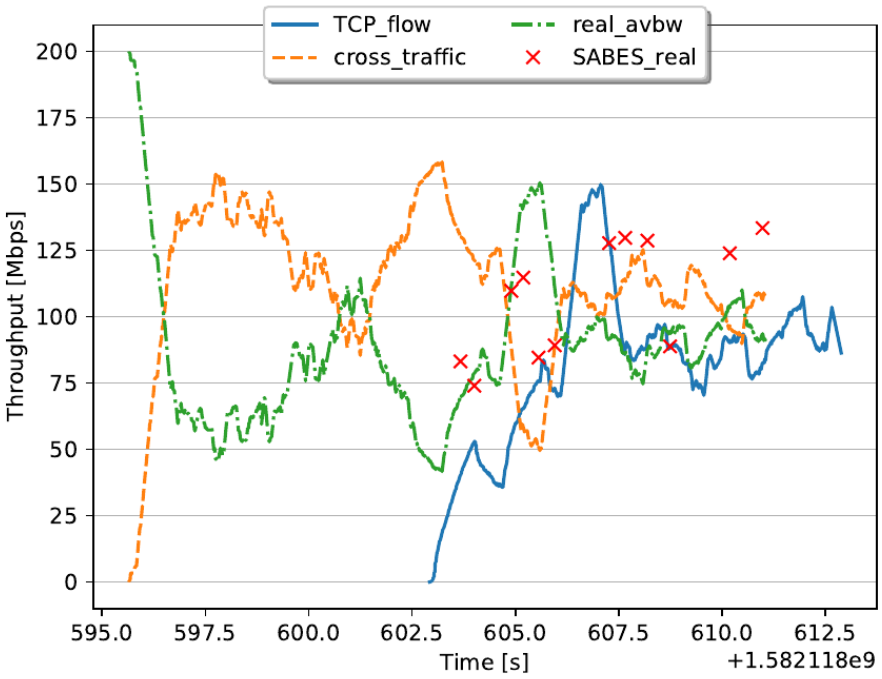

Figure 5.4. Available Bandwidth real-time estimation (fig.1)

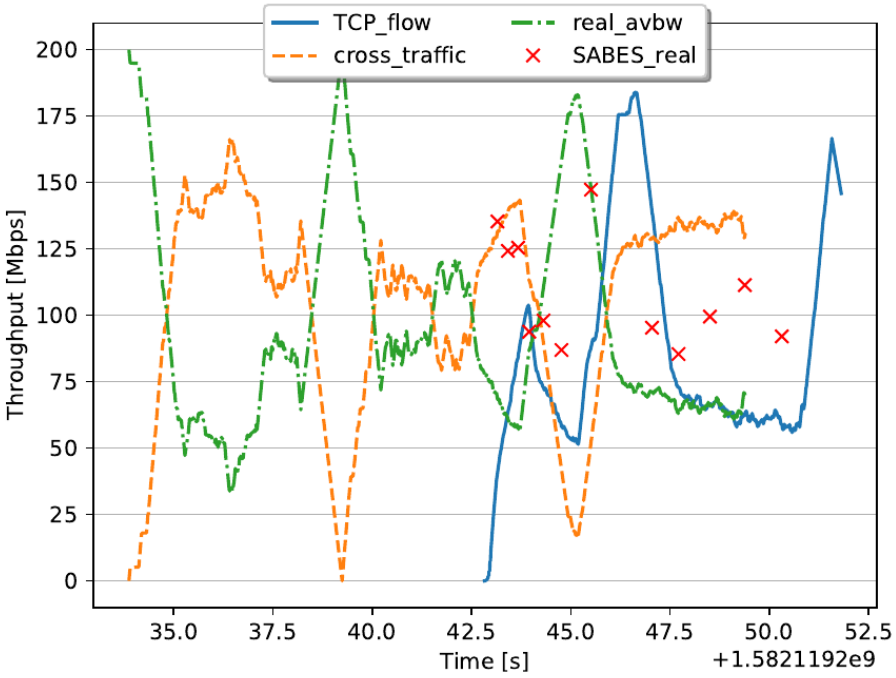

Figure 5.5. Available Bandwidth real-time estimation (fig.2)

Chapter 6

Conclusions and future work

By the measurements and estimations obtained, the method studied and used during this Thesis Project at Clevernet is surely valid, promising and innovative.
The Available Bandwidth real-time estimation method is pretty good, constituting a reliable starting point for the final implementation of the controller to be deployed on the edge routers in charge of modulating the TCP receive window.

Bibliography

[1] Francesco Ciaccia, Oriol Arcas-Abella, Diego Montero, Ivan Romero, Rodolfo Milito, René Serral-Graciá, Mario Nemirovsky, *Improving TCP Performance and Reducing, Self-Induced Congestion with Receive Window Modulation*, date, Pages 1-2

[2] Brian Storti, *TCP Flow Control*, June 30, 2017, https://www.brianstorti.com/tcp-flow-control/

[3] Wikipedia, *Transmission_Control_Protocol*, https://it.wikipedia.org/wiki/Transmission_Control_Protocol

[4] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, Van Jacobson, Amin Vahdat, *TCP BBR congestion control comes to GCP – your Internet just got faster*, July 20, 2017, https://cloud.google.com/blog/products/gcp/tcp-bbr-congestion-control-comes-to-gcp-your-internet-just-got-faster

[5] Alexey N. Kuznetsov, *tc-tbf*, https://linux.die.net/man/8/tc-tbf

[6] Matt Fleming, *A thorough introduction to eBPF*, December 2, 2017, https://lwn.net/Articles/740157/

[7] Brendan Gregg, *Linux Extended BPF (eBPF) Tracing Tools*, 2019, http://www.brendangregg.com/ebpf.html

[8] Alberto Dominguez, *Why does CUBIC take us back to TCP congestion control?*, July 10, 2019, https://pandorafms.com/blog/tcp-congestion-control/

[9] Alessio Botta, Walter de Donato, Alberto Dainotti, Stefano Avallone, Antonio Pescapé, *D-ITG 2.8.1 Manual*, October 28, 2013, p. 3, http://www.grid.unina.it/software/ITG/manual/D-ITG-2.8.1-manual.pdf

[10] Wikipedia, *tcpdump*, https://en.wikipedia.org/wiki/Tcpdump

[11] Wikipedia, *pcap*, https://en.wikipedia.org/wiki/Pcap

[12] Fabio Ludovici, Hagen Paul Pfeifer, *NetEm - Network Emulator*, November 25, 2011, http://man7.org/linux/man-pages/man8/tc-netem.8.html

[13] Wikipedia, *Interrupt coalescing*, https://en.wikipedia.org/wiki/Interrupt_coalescing

[14] da completare, *Generic Receive Offload*, https://doc.dpdk.org/guides-18.05/prog_guide/generic_receive_offload_lib.html