UNIVERSITE CATHOLIQUE DE
LOUVAIN

ECOLE POLYTECHNIQUE DE
LOUVAIN

Evolution of the Linux TCP stack: a simulation study

Supervisor: Olivier Bonaventure
Readers: Ramin Sadre, Fabien Duchêne

Thesis submitted for the Master's degree
in computer science and engineering (120 credits)
options: networking and security
by Esteban Sastre Ferrández

Louvain-la-Neuve
Academic year 2014-2015

"My laughter is my sword, and my joy is my shield."

Martin Lutero.

Abstract

TCP (Transport Control Protocol) is widely known as a communication protocol between devices that guarantees a reliable, ordered and error-checked exchange of information flow between them. As research progresses, changes in it's design and architecture are made, new features are added or modified, etc.. This translates into the existence of a large variety of different implementations of the same. Every new *Linux Kernel* version comes on the heels of representative changes inside TCP that may incur an enhancement or a disadvantage when it comes to use one or another in certain specific network scenario. Measuring performance of all distinct versions of the protocol turns into an impossible task to carry out in the ample variety of heterogeneous real environments. This thesis provides a set of simulations of different TCP implementations under diverse traffic scenarios, all performed in several network topologies considered representative to a range of real cases. To achieve this goal, simulations are done within the ns-3 discrete event network simulator along with the DCE(Direct Code Execution) framework. Obviously taking into account every potential scenario is impossible. Hence, what this work offers is a representative subset of tests that may be useful to later researchers/sysadmins as a reference when it comes to deploy one or other version in a real environment.

Acknowledgements

I wish to express my sincere thanks to Olivier Bonaventure, my promoter, for suggesting me the topic of the thesis, which I had no previous experience on it and gradually grew interest on me. I am also grateful to Fabien Duchêne, lecturer, in the Department of INL (Internet Networking Lab, Lovain-la-Neuve, Belgium). I highly appreciate his insight and previous experience with the tools I used to perform the experiments. Also to Ramin Sadre, who kindly accepted to take part of the jury for this thesis without asking him with too much time in advance.

I take this opportunity to express gratitude to Francisco Novoa, my co-promoter in Spain, who went through the hassle to care about my work during my period abroad. I also thank my parents for the unceasing encouragement, support and attention.

Finally, I too pace my sense of gratitute to all new people met in this new country and grew support on me even in my lowest states of mind.

Contents

List of Figures

Abbreviations

MTU	**M**aximum **T**ransmission **U**nit
TCP	**T**ransport **C**ontrol **P**rotocol
RTT	**R**ound-**T**rip delay **T**ime
MSS	**M**aximum-**S**egment**S**ize
HTTP	**H**yper-**T**ext **T**ransfer **P**rotocol
PRNG	**P**seudo-**R**andom **N**umber **G**enerator
IP	**I**nternet **P**rotocol

Dedicated to my family, the most important thing in the world.

Chapter 1

Introduction

Progress and research in computer networks have come to a point where the quantity of existing new technologies is overwhelming. Not only new software or hardware products are developed, but also new network topologies are designed and deployed so all jointly satisfies the actual market needs.

That is why by the time to decide if one or another network technology is going to be deployed in a real environment, specially in a critical one, it becomes necessary to carry out a complete and meticulous study on those technologies behaviour when used in those different environments to verify if it was the right choice.

Computer network simulation provides a powerful tool when it comes to measuring performance of a wide variety of settings conformed by different devices, channels, communication devices, and programs using those ecosystems to transmit information between devices.

In this thesis a set of performance tests is presented. Those are executed over the distinct implementations of the TCP protocol throughout all different Linux Kernel versions. New kernel releases usually are accompanied by significant changes in TCP/IP stack of side-enhancements in the form of kernel extensions which are translated into a network performance advance. Despite the fact those changes are not incorporated in every new version, for certain particular releases changes are introduced so they constitute a big impact in protocol's amelioration. It is in those versions where it is worthy check their behaviour's quality in a different range of scenarios.

Obviously, executing tests for all possible combinations of topologies, devices, channels and models (*LTE*, *WiMAX*, *WiFi*, etc..) is impossible. Work will be narrowed down to a subset of cases that represent generic examples of similar scenarios. Therefore a representative example of traffic exchange for well-known cases, encompassing either technologies or traffic scenarios, will be provided.

Actual studies and performance comparisons on different TCP implementation focus

specially in performance collation between different flow congestion control algorithms (*cubic*, *reno*, ..) for a same kernel over one or several topologies. They are also common papers or articles where for a given network topology and an established traffic flow between two devices, several parameters in the TCP/IP stack are adjusted and then the variations on protocol performance are observed for that particular scenario.
The remainder of the thesis will be structured in this way: first section is dedicated to TCP: a brief overview on how the protocol works is explained along with the roadmap of the evolution of the protocol. Also, an introduction on how TCP congestion control mechanism works is mentioned, and in last place, important TCP extensions that directly affect performance are enumerated and explained. Second section is dedicated to enumerate all the tools and technologies that will be used during the development of the thesis. Next one focuses on explaining in depth the main characteristics of the *ns3* simulator that will be used for the simulations in conjunction with the *dce* framework, mentioning it's particularities. Fifth section is entirely dedicated to analyze and extract conclusions on the simulations performed, and last section will be the final conclusions on the overall experiments and results, providing general insight about the work done.

Thesis written by —
Esteban Sastre Ferrández

Chapter 2

TCP

2.1 An overview of TCP

The Transport Control Protocol (TCP) stands as one of the main pieces that take part on the *Internet Protocol Suite*. It is connection oriented and provides a reliable, in-order, unduplicate delivery of data, start and termination of connections, congestion control and proper indication of urgent data. TCP is by far the main protocol being used by the majority of applications of the internet. *Email*, *World Wide Web*, *File Transfer*, *Secure Shell* or *Peer-to-Peer* are fair examples of ubiquity. It was first presented in 1974, the main paper can be found here[RFC793].

TCP acts as a communication interface at an intermediate level between an application program and the Internet Protocol. It provides connectivity at the Transport Layer, so any applications running on top don't have to be acquainted of the undergoing mechanisms that take place to do so. IP provides an unreliable service, so it is TCP's duty to provide a byte-stream end-to-end flow control. To that end, it uses checksums, sequence numbers, windows and acknowledgement of data received.

A typical TCP protocol operation is divided intro three phases: the *connection establishment*, in which two end hosts start a new connection via the so called *3-way handshake*, then the *data transfer* phase, and after the data transmission has finished, the last step, *connection termination*. Once the connection is established between hosts data is exchanged in data units called segments. The windows limit the amount of data that one sender can send in consecutive segments. TCP keeps track of the individual units, because sometimes can be divided into several small pieces for efficient routing through the network. When those segments are lost, in order to ensure reliability, TCP retransmits after a timeout. When segments are received, in order to provide error detection capabilities the acknowledgement numbers and other flags carried in the TCP segment

are used to decide acceptability and also to take decisions into which actions must be carried on.

Finally, the other main aspect of TCP is congestion control. Data from applications in end hosts is passed to TCP, then encapsulated by IP and then to the network. Those packets are transported over routers, and switches, devices in charge to take routing decisions accordingly to those packets and their own internal behaviour. These routers and switches have software buffers in order to store packets before sending them over the network, and when there is an extensive agglomeration of packets in those devices, buffer overflow and packets are discarded. This packet losses serve to detect network congestion and thus take the corresponding actions. TCP uses a number of techniques and mechanisms to avoid this situation, these mechanisms control the rate of data entering the network, keeping the data flow below a rate that would trigger collapse. Acknowledgments for data sent, or lack of acknowledgments, are used by senders to infer network conditions between the TCP sender and receiver. Along with timers, TCP senders and receivers can alter the behavior of the flow of data.

2.2 Evolution of TCP

All over it's history, TCP has been sometimes almost reimplemented accross several platforms and operating systems, modified and extended. As it has been evolving, many RFC's (Request For Comments) have become standard in definition and implementation. RFC's constitute a series of publications of the *Internet Engineering Task Foce (IETF)* that describe diverse aspects of Internet's workings and other computer networks, aswell as protocols, procedures, and comments and ideas about them. Parallel, several distinct experimental features under development have been published in various RFC's, and although some of them are not standard or widely implemented, many others are slowly becoming strongly suggested to be incorporated to the main design of it. This section provides an overview on the evolution the protocol from it's very core features until the lately proposed extensions. A detailed version of the roadmap on TCP can be checked in [RFC7414]. This part will only cover and mention those ones directly related with TCP/IP and documentation that describes mechanisms that affect performance, staying out those that focus primarily on security. Another thing to be taken into account is that the set of documents mentioned fall into one of these three categories: core documents, which constitute an integral part of the protocol, strongly suggested extensions, and experimental (not widely deployed, does not mean explicitly unstable) ones. The majority of the RFC's fit into the core or strongly suggested categories, but almost can be considered both as core features due to the wide knowledge and deployment of them. Even those extensions mentioned considered experimental, by

the time of the writing of this thesis (*Linux Kernel 4.0* about to be released) are already extensively implemented and deployed.

In **1981** [RFC793] was published. It constituted the main Transmission Control Protocol Specification. This first specification enumerated the main pillars of it's inner workings: the packet format, state machine, event processing, and semantics for data transmission, reliability, flow control, multiplexing and acknowledgements. Some parts of that RFC are irrelevant nowadays as those mechanisms changed, however, the majority of the draft still accurately describes modern versions.

In **1990**, the [RFC1191] Path MTU Discovery added a new technique for dynamically discovering the *Maximum Transmission Unit (MTU)* of an arbitrary internet path. All diferent devices over the internet have different MTUs supported, so, is TCP duty to perform the segmentation of packets that are after encapsulated over IP to be sent over the network in case that they exceed the MTU of the devices they have to go through. This is important because TCP performance may be affected if the sender's maximum window size is not an exact multiple of the segment size in use.

In **1992**, the main RFC, [RFC1323]: TCP Extensions for High Performance comes to light, where window scaling, timestamps, and protection against wrapped sequence numbers are presented for the first time.

Six years later in **1996** the [RFC2018]: TCP Selective Acknowledgment Options draft comes to light, providing one well known data transfer enhancement mechanisms the protocol uses nowadays: *Selective Acknowledgements (SACK)*. This mechanism allows a receiver to only acknowledge received segments so only the ones that have not yet arrived are going to be retransmitted.

Within the same year of the Path MTU Discovery paper, and then in the late nineties (**1997**) two performance related RFC's are published: [RFC1144], and [RFC2140]. The first one, Compressing TCP/IP Headers for Low-Speed Serial Links is quite self-explanatory, and the latter, TCP Control Block Interdependence states how TCP connections under the same endpoints should share information, such as congestion control state.

At the beginning of the year **2000**, a new [RFC2883] is released, adding a new twist to the *SACK* mechanism: the *Duplicate SACK (DSACK)*. It enables use of the SACK option to acknowledge duplicate packets. With this extension the sender is able to infer the order of packets received at the receiver and, therefore, to infer when it has unnecessarily retransmitted a packet.

Along with it, the [RFC2861] comes out, suggesting reducing the congestion window over time when no packets are flowing. Next year, **2001** makes TCP reach two milestones with the RFC's [RFC3168] and [RFC3042]. The first one was The Addition of Explicit Congestion Notification (ECN) to IP, where a mean for end hosts to detect

congestion before congested routers are forced to discard packets was defined. The second one: Enhancing TCP's Loss Recovery Using Limited Transmit, provided the idea of *Limited Transmit*, that can be used to more effectively recover lost segments when a connection's congestion window is small, or when a large number of segments are lost in a single transmission window. It is a less lenient version of Early Retransmit, defined in later RFC's.

In **2002**, a minor but important draft (Increasing TCP's Initial Window) [RFC3390] specified an increase in the permitted initial window for TCP from one segment to three or four segments during the slow start phase, depending on the segment size.

The year **2003** brought only two significant documents into the field: [RFC3649] (High-Speed TCP for Large Congestion Windows) and [RFC3522] (The Eifel Detection Algorithm for TCP). First one is a new congestion control algorithm and the other is an algorithm designed to detect if TCP has entered Loss Recovery.

During **2004**, only two lesser documents were presented that could affect performance and are not worth being mentioned. In **2005** there is also only one draft presented refining the behaviour for the Eifel algorithm in the response part. **2006** Comes also with no major renewals or additions to improve the protocol.

In **2007** [RFC4821] is published. "Packetization Layer Path MTU Discovery" provides a method to probe an internet path with progressively larger packets, which can result in a pretty important performance enhancement.

Year **2009** stands as a really important one for TCP. Multiple papers with key improvements came to light. The TCP Congestion Control [RFC5681], defines congestion avoidance and control mechanism for TCP implementations. TCP congestion control was not yet specified in the main RFC, so is it in this document where well-known congestion control techniques are thoroughly explained: slow start, congestion avoidance, fast retransmit, and fast recovery. The "TCP Reno" fast recovery congestion control algorithm is first exposed in **1988**.

Other very important piece of work is the publication of the Forward RTO-Recovery (F-RTO) [RFC5682], which states a new algorithm for detecting spurious retransmission timeouts. The third main document is [RFC5482] (TCP User Timeout Option). The user timeout controls how long transmitted data may remain unack'ed before a connection is forcefully closed. It allows one end of the connection to advertise it's current user timeout value, so the other end of the connection can see it in order to find synchronization. It works as a per-connection parameter.

Last but not least, the mention of the experimental [RFC5562] describes an approach that allows TCP to add ECN capabilities to SYN/ACK packets, permitting them to alert about congestion instead of the RTT that would be lost due to the packet drop. In year **2010** the document where *Early Retransmit* mechanism was explained is published: [RFC5827]. That new technique allows to recover lost segments in a connection when

the congestion window is small, therefore reducing the number of duplicate acknowledgements required to trigger fast retransmit to recover segment losses without waiting for a lengthy retransmission timeout.
Between **2011** and **2012** is worth mentioning [RFC6093], and [RFC6582]. First one analyzes how current stacks process urgent indications and strongly suggest the termination of it's use. Second one, The NewReno Modification to TCP's Fast Recovery Algorithm shows the new congestion control algorithm NewReno, which turns to be a modification of Reno but this one is served by partial acknowledgements to infer the new data that must be send and no SACK is available. NewReno can make a big performance difference when multiple segments are lost in a single window of data. And, finally, [RFC6675] (A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP). This last one provides a conservative loss recovery algorithm based on the use of the SACK option.
Then, in **2013** an experimental paper (Proportional Rate Reduction for TCP) comes out ([RFC6937]), which stands as an apparent improvement to Fast Recovery.
Two important RFC's get published in **2014**: [RFC7323] (TCP Extensions for High Performance), which only complements the main TCP Extensions for High performance in **1990** ([RFC1323]); and [RFC7413], TCP Fast Open, describing the mechanism that allows to carry data in the SYN and SYN/ACK packets used by the receiver at the beginning of the connection.

2.3 TCP Congestion Control algorithms

Congestion control as Wikipedia states, comprehends several mechanisms implemented in the protocol that control the rate of data entering the network, keeping the data flow below a rate that would trigger collapse. They can be seen as algorithms that perform each one a different kind of strategy to try to maintain a continuous flow of information between two TCP ends without causing the network to lose packets. They also try to protect against an overload or blocking in the network.
It is out of the scope of this thesis to explain the behaviour and inner workings of every existing TCP congestion control algorithm, however during it's development it will be necessary to explain some of those algorithm's specific strategy at some point in order to understand the results we are looking at.
However, in *Congestion Control* there are several mechanisms that all different TCP implementations share, being adjusted by each of them in order to perform in different ways and following different strategies: *Slow Start*, *Congestion Avoidance*, *Fast Retrasmit* and *Fast Recovery*. Those ones constitute the common trunk of TCP congestion control, with changes in every implementation.

2.3.1 Slow Start

It constitutes the first phase of the Congestion Control strategy. The algorithm is simple: at first, it makes the sender to start with an initial congestion window size (**cwnd**) of 1 or 2. Then, with every received ACK the cwnd will be doubled in size (or following another strategy depending on the TCP implementation). Transmission rate will keep increasing until either a loss is detected, or the cwnd size surpasses the receiver's advertised window (**rwnd**), of an arbitrary control value called slow start threshold (**ssthresh**) is reached. If the stopping factor was a loss, then the cwnd is reduced accordingly to the current congestion control algorithm (i.e. cubic). Some algorithms modify the Slow Start rule by making cwnd grow following a different rule than the previously exposed.

2.3.2 Congestion Avoidance

Once a loss is detected, the *Congestion Avoidance* phase is the one in charge to "cut" the cwnd in order to reduce network load and recover stable state. It is in this phase where different strategies for reducing and increasing the congestion window come into play, depending on the congestion control algorithm used.

2.3.3 Fast Retransmit

Fast Retransmit acts as an improvement to save one RTT in TCP. It works as follows: the TCP sender waits normally for three duplicate ACK's to consider a segment to be lost. The sender can then assume that the segment with the next higher sequence number was also dropped, so it will presume it was also dropped and retransmit it before waiting for it's timeout.

2.3.4 Fast Recovery

After the three duplicate ACK's are received, TCP goes into this algorithm and not Slow Start in order to provide a higher throughput. It is implemented together with the Fast Retransmit algorithm.
Once the three duplicate ACK's are received in a row:

1. Set *sshthresh* to half the current window.
2. Retransmit the missing segment.
3. *cwnd=sshthresh+3*

4. Each time the same duplicate ACK arrives, set *cwnd = cwnd+1*. Transmit a new packet if allowed by cwnd. If non-duplicate ACK arrives, then *cwnd = sshthresh* and continue with congestion avoidance.

2.4 TCP Performance Extensions

This mechanisms shape the main group of performance extensions used in the Linux Kernel. Most of them are already present from the first kernel version we will be testing in this thesis *2.6.36*, and some of them are added in further versions. Still, all of them may suppose a significant performance improvement when implemented in the TCP stack.

2.4.1 Selective Acknowledgements (SACK)

Figure 2.1 gives us insight on how *SACK* extension works. When multiple segments are lost in an RTT (a window of data), TCP may experience poor performance. The sender can only learn about one lost segment each round-trip time or retransmit those packets. An aggressive sender could retransmit packets earlier, but those packets may have been already successfully received. The only mechanism it has to infere losses is by waiting to receive three duplicate ACK's.
The *Selective Acknowledgement (SACK)* mechanism is a strategy designed to correct and improve this behaviour in the face of multiple dropped segments. It consists basically that the receiver part can inform the sender about which data was properly received by using the so-called SACK blocks. Then the sender can then retransmit only the missing data segments.
The SACK option is included in the TCP header and both ends have to support it to correctly work, and it is enabled by default since the *Linux Kernel* version 2.2.

There is also a mechanism for improving congestion control that uses the potency of SACK, called *Forward Acknowledgement (FACK)*[15].
Another mechanism that takes advantage of SACK's capabilities is the one specified in the [RFC2883], called *Duplicate-SACK (D-SACK)*. The original draft explaining SACK in [RFC2018] does not cover the use of SACK option when duplicate segments are received. This new document suggests that when duplicate packets are received, the first block of the SACK option field can be used to report the sequence numbers of the packet that activated the acknowledgement.
This new option allows the sender to deduce if it has unnecessarily retransmitted a

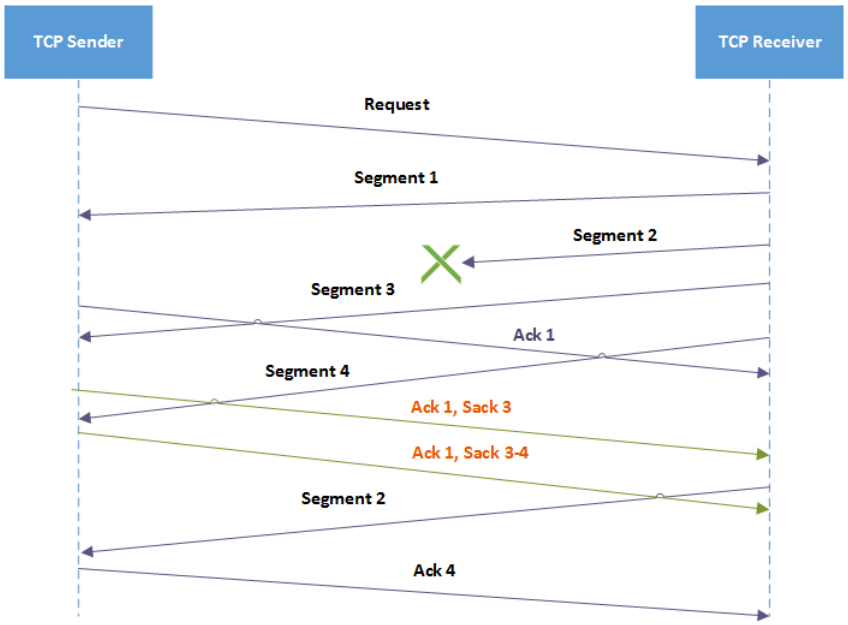

FIGURE 2.1: TCP SACK

packet. This new capability can serve to enhance more robust operation in situations like packet replications, early retransmit timeouts, ack loss or reordered packets.
The D-SACK option is compatible with any implementation of TCP+SACK as it doesn't require new negotiation between ends. It means that one end could send D-SACK blocks even if the other end does not understand this new extension, by processing them as normal SACK ones.

2.4.2 Early Retransmit

Before explaining what is *Early Transmit*, it's mandatory to first mention a mechanism that served as an earlier version of the aforementioned: *Limited Transmit* [RFC3042].
In situations where the congestion window at the TCP sender is small, there has been observed that loss recovery strategies were not working well, either because there was only a limited amount of data to send, or by the limit imposed by the receiver's advertised window.
Once TCP detects a missing segment it will enter loss recovery by using one of the two methods: on one hand if an ACK is not received by the sender in a certain amount of time, a RTO will fire up and segment will be retransmitted. On the other hand whenever three duplicate ACK's arrive at the sender it will trigger "Fast Retransmit" mechanism and resend the segment too.
The *Limited Transmit* algorithm sends a new data segment in response to each of the first two duplicate ACK's that arrive at the sender if two conditions are fulfilled: receiver's advertised window allows, and outstanding data would be within the congestion

window plus two segments.
Those segments increase the probability that TCP can recover from a single lost segment using the Fast Retransmit algorithm rather than waiting for the Retransmission Timeout. Limited Transmit can be used with or without SACK option enabled. Summarizing, it improves throughput when a sender cannot receive three DUPACKs due to the congestion window being too small.
Early Retransmit[RFC5827] acts as a refinement or extension of Limited Transmit capabilities. By sending those two new segments with Limited Transmit, the sender is attempting to create additional duplicate ACKs so Fast Retransmit will be triggered before the RTO expires and thus saving up time. But what would happen in a case when previously unsent data is not available for transmission (or in other words, there is no more data to send), or the receiver's advertised window will not allow the sender to transmit new packets? There is where this new mechanism comes into play.
The algorithm searches to lower the threshold for triggering Fast Retransmit when the amount of outstanding data is small and when no previously unsent data can be transmitted. It does so by a simple rule: if the congestion window is less that four times the MSS and the sender cannot send new data via Limited Transmit, then it triggers Fast Retransmit on *cwnd-1* duplicate ACKs.
As it's predecessor, Early Retransmit can be used with or without SACK option enabled.

2.4.3 Tail-Loss Probe (TLP)

The Transmission Control Protocol (TCP) has two methods for recovering lost segments. First, the fast retransmit algorithm relies on incoming duplicate acknowledgments (ACKs), which indicate that the receiver is missing some data. After a required number of duplicate ACKs have arrived at the sender, it retransmits the first unacknowledged segment and continues with a loss recovery algorithm. The second method are Retransmission Timeouts (RTO). They stand as a fixed amount of time TCP will wait when it has sent a segment that has not received an acknowledgement yet. Once this timeout ends the sender will retransmit the already assumed lost segment again. They can consist on a harmful threat to application performance, being specially pernicious in terms of latency increase for short transfers such as Web transactions, where timeouts themselves can often take longer than all of the remaining transaction. In terms of performance, lossy responses last ten times longer than lossless ones. 30% of losses are recovered by TCP's Fast Recovery algorithm, whereas the remaining 70% by timeouts. As stated before, that originates a problem when working with short transactions such as web traffic, as stated in original draft: [drafttlp], shown in a more friendly way in Google's presentation. *Tail Loss Probe* [drafttlp] is an algorithm to invoke fast recovery

for tail losses that would otherwise be only recoverable through timeouts. Tail losses are those that occur in the last segment or segments of the transaction.

A question comes up to mind: Can a sender recover tail losses through fast recovery and avoid long retransmission timeouts? That is TLP's goal, and achieves so by converting retransmission timeouts into fast recovery, by sending probe segments to trigger duplicate ACKs with the intent of invoking fast recovery more quickly than an RTO at the end of a transaction. The transmitted packet can be either new or a retransmission. If it happens to be a tail loss, the ACK from the loss probe will trigger fast recovery immediately, therefore avoiding a costly RTO. This is better seen in a graphical form in next Figure 2.2

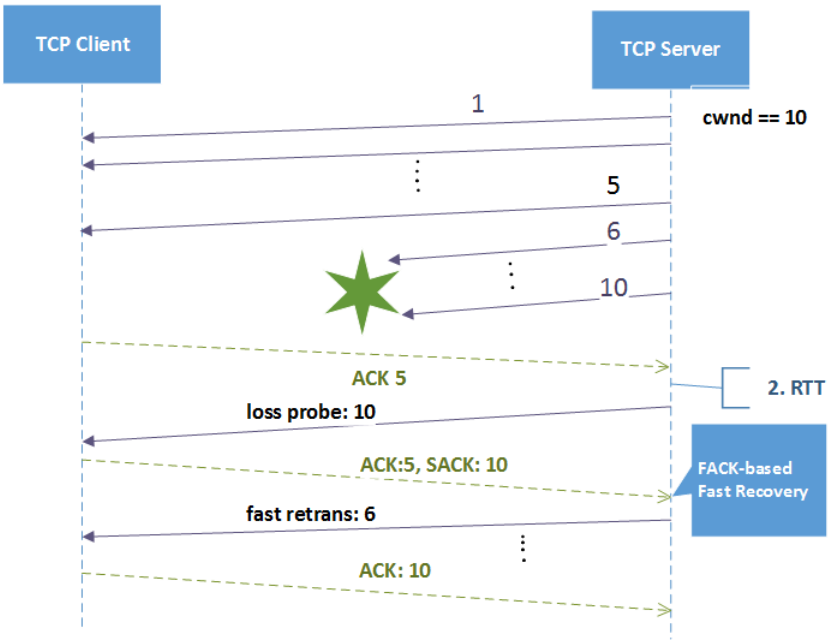

FIGURE 2.2: Tail Loss Probes

2.4.4 TCP Forward-RTO

As previous subsection explained, TCP has two main mechanisms to detect losses: the three duplicate ACK's received in the sender side of a connection, or the expiration of the RTO timer, both of them trigger the retransmission of the first unacknowledged segment. About what concerns the RTO timer, it has been proven that it can expire spuriously and cause unnecessary retransmissions when no segments have been lost. There are several reasons that may cause spurious retransmissions. In mobile networks, there are sometimes sudden delay spikes because of actions taken during a hand-off, or in low-bandwidth link with arrival of competing traffic, can increase the round-trip time. There are other spurious retransmission detection and recovery algorithms such as Eifel [RFC3522], or DSACK-based algorithms [RFC2883]. While Eifel makes use of the TCP Timestamps and DSACK-algorithms require that both ends support SACK and have the DSACK extension too, F-RTO does not require any TCP options to operate. The algorithm only attempts to detect and avoid RTO-expiration caused retransmissions, while the others also contemplate unnecessary retransmissions caused by other events, for example packet reordering.
How does F-RTO work then? Once the retransmission timeout has expired, the sender retransmits the first unacknowledged segment. Then, it tries to transmit new data, for the first ACK that arrives after the timeout. If the second ACK arrives and acknowledges data that was not retransmitted, the timeout is therefore considered spurious and exits RTO recovery. Still, if any of those two ACK's is a duplicate one, you cannot infere if it is really a spurious timeout and therefore the sender will retransmit the unacknowledged segments in slow start. Forward-RTO is active since the *Linux Kernel version 2.4.0*

2.4.5 Small Queues

Small Queues [4], and [5] is a mechanism first introduced in the *Linux Kernel 3.6.0* that allows TCP to fight against the bufferbloat problem. Bufferbloats is a situation that arises mainly in packet-switched networks, where excess buffering of packets provokes high latency and packet delay variation, as well as affecting overall network throughput. It is an issue caused by router and switch vendors, who make incorrect assumptions about buffering packets for too long when they should drop them. Bufferbloat constitutes a real problem only when the buffer is actually used, if the link they are buffering is the one with the bottleneck being congested, causing the packets to stay too long there.
What this improvement does is to make the TCP stack aware of when packets leave the system. It limits the amount of data that can be queued for transmission by any given socket regardless of where the data is queued. A new tcp entry is added in `/proc/`

`sys/net/ipv4/tcp_limit_output_bytes` and a default value of **128KB** is established. This entry is a configurable limit of bytes in transit in lower layers, after that limit, packets are kept at the TCP layer.

2.4.6 Byte Queue Limits (BQL)

Byte queue limits [10] and [11] are a mechanism to limit the size of the transmit hardware queue on a NIC by number of bytes. Packets spend enough time enqueued inside the stack, and when a packet gets to a NIC it is enqueued again. The objective of this technique is to set the limit to be the minimum needed to prevent starvation between successive transmissions to the hardware. Putting it simple, it reduces unnecesary queuing in the NIC by setting a limit, and then pushes packets to the network stack.
This mechanism is available since *Linux Kernel 3.3*.

2.4.7 Round-Trip Time Measuring (RTTM)

The *Round-Trip Time Measuring (RTTM)* mentioned in [RFC1323] stands as one of the extensions for improving TCP performance that has a direct impact on some speed-related and bottleneck provoking issues. RTTM relies heavily in other mechanism previously mentioned in that rfc called *TCP Timestamps option*.
This previous one adds a new option on transmitted segments that can help to determine in which order packets were sent. Two values are used: a sender timestamp and an echo reply timestamp, both are not aligned to the system clock and start at some random value. This technique is used in other algorithms such as *Protection Against Wrapped Sequence numbers (PAWS)* or the *Eifel* detection algorithm.
RTTM serves to measure the round-trip time of every packet acknowledged. It requires that all measured segments use the timestamps option. The mechanism it uses to do so is simple, to values are added to the packet: **TSval**, which is obtained from a virtual timestamp clock, and **TSecr**, which is nothing else but the previous TSval value echoed back in the reverse direction. The difference between a received TSecr and the current timestamp clock is therefore the RTT measurement. This following scheme extracted from the original RFC illustrates how does this mechanism work: TSval and TSecr are symbolic values to illustrate the timestamp value and echoed response.

```
TCP  A                                              TCP B

                    <A,TSval=1,TSecr=120> ------>

          <---- <ACK(A),TSval=127,TSecr=1>

                    <B,TSval=5,TSecr=127> ------>

          <---- <ACK(B),TSval=131,TSecr=5>
```

2.4.8 TCP Fast Open

The *TCP Fast Open (TFO)* technique appeared first in **2011** [draftfastopen], when several Google developers proposed a draft to reduce latency by streamlining the process of opening a TCP session. It's main goal is to speed up the process of opening succesive connections between to endpoints. This extension was first included in *Linux Kernel 3.6* in client-side and in *Linux Kernel 3.7* in server end.

Fast Open is based on the idea that most of nowaday's internet's bulk traffic is characterized by relatively short-lived data streams. When a browser opens a web page for example, it creates multiple brief TCP connections to transfer relatively small data (such as web page's elements: small graphics, javascript, html code..). As the performance cost associated with establishing connections continuously, browsers will often try to keep unused connections open in case of being utilized subsequently. This, however, may result in a myriad of idle TCP connections doing nothing but consuming system's resources.

Normal TCP connections are performed in the usual way: client sends SYN packet, server responds with a SYN/ACK one and the client sends a later ACK to properly finish the negotiation. At this time, the client is not able to send actual data until the last ACK packet is sent, hence losing one entire RTT that can be avoided by using TFO. The main idea behind this technique is to be able to send data directly in the first SYN packet. When two clients are connected in first place, a TFO-enabled client will send a connection request, then the server will create what is called a *TFO cookie*, which is cryptographically generated and then sent to the client to be stored there. That cookie among other things contains the encrypted IP of the client. After the normal TCP negotiation, along with the generation of the TFO cookie, now the client can use it to avoid repeating the handshake in future connections. If a client now needs to reconnect to the server after closing a previous connection, now it sends the TFO cookie in the initial SYN request along with the initial payload data, the server then validates the

cookie and ensures that the encrypted IP address matches the requesting client's one. If everything is ok, the server can return data before receiving an ACK from the client, and thus avoiding a complete RTT. Next two figures represent first the connection setup and generation of TFO cookie, and then a TCP connection establishment once the cookie has been generated and sending of payload data in the beginning.

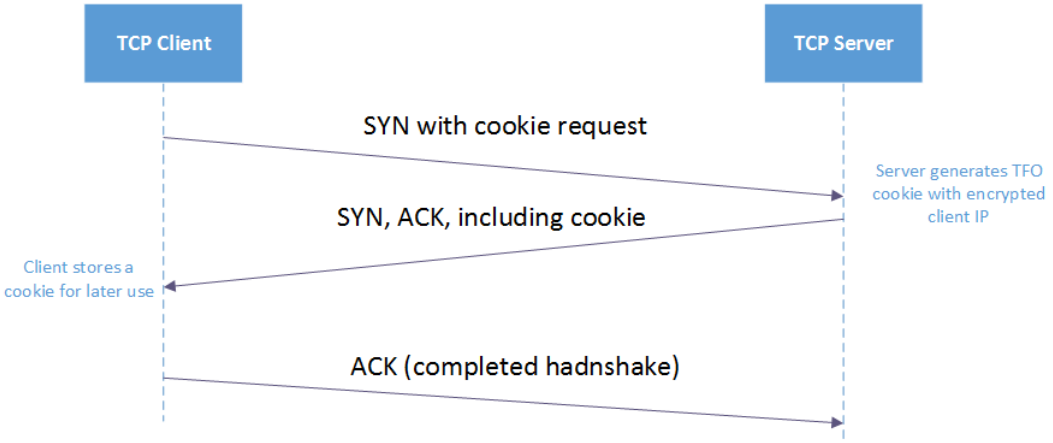

FIGURE 2.3: TCP Fast Open Init

As we can see, the first time the client connects to the server, the TFO cookie is generated in the server and stored in the client.

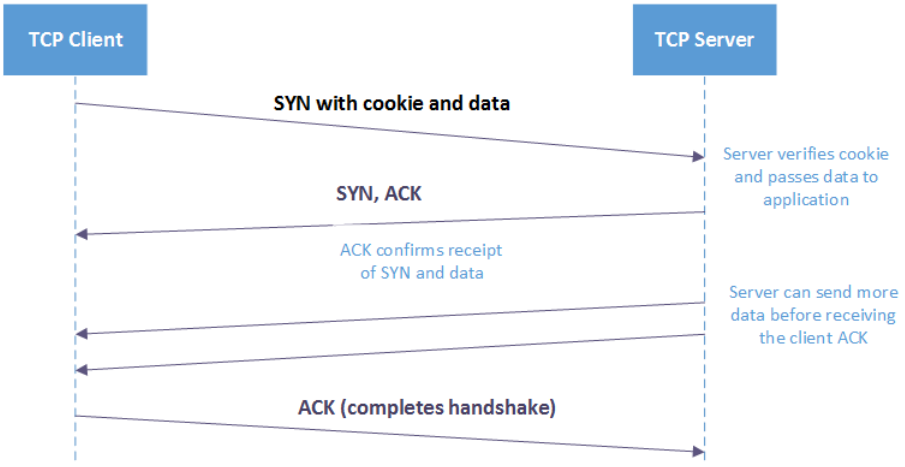

FIGURE 2.4: TCP Fast Open Connection

And then, the next time the client needs to reopen the connection, it authenticates itself with the cookie and starts sending data from the very beginning.

2.4.9 TCP Corking

This mechanism has been active in the Linux kernel sin version *2.2*. It is a socket option, enabled by activating the **TCP_CORK** flag when defining it's options:

```
1   int state = 1;
2   setsockopt(fd, IPPROTO_TCP, TCP_CORK, &state, sizeof(state));
```

When set, it basically prevents the kernel to send half-sized packets. Only full packets are sent, until TCP_CORK option is again removed. If there is partially queued data when the flag is removed, it is automatically sent. It is useful for throughput optimization. It can be used in combination with **TCP_NODELAY** option, that deactivates Nagle's Algorithm (wait for one end to acknowledge the previously sent packet before sending any partial packets).

2.4.10 Proportional Rate Reduction (PRR)

Proportional Rate Reduction was first presented in **2011** as an internet draft by three Google engineers [draftprr] and then accepted as an official standard in [RFC6937], in **2013**. PRR stands as an alternative to the two loss-recovery algorithms Fast-Recovery and Rate-Halving. Those algorithms have their own estimation on the amount of data that must be sent by TCP during loss recovery.
In a nutshell, Fast Recovery will wait for half of ACKs to pass before sending data, and Rate-Halving will send data on alternate ACKs during recovery, such that after 1 RTT the cwnd has been halved. It also often allows the congestion window to fall further than necessary, increasing the risk of timeouts if there are additional losses in the network.
PRR tries to tackle those limitations by aiming at two main goals: reducing timeouts by avoiding excessive window reductions, and converge to cwnd chosen by congestion control by the end of recovery, so it will be as close as possible to ssthresh value determined by the congestion control algorithm.
It works in two steps: at first, it will calculate an estimation for the amount of data still in flight plus an additional calculation on how (according to the congestion control algorithm in use) the cwnd should be. If the amount of data in the pipeline is less than the target cwnd, it enters Slow Start. Otherwise, if the amount of data in flight is at least as large as the new congestion window, an algorithm like Rate-Halving is used but reduction is relative to the congestion window, rather than the half cut of Rate-Halving.
This performance improvement mechanism has been integral part of the *Linux Kernel* since *version 3.2*

2.4.11 Proportional Integral Controller Enhanced (PIE)

From the Linux Kernel version *3.14*, the Proportional Integral controller Enhanced (PIE)(see [draftpie]) was added as a new mechanism in order to fight the bufferbloat problem. It aims to control the average queuing latency and achieve high link utilization under various congestion situations. It is simple to implement and does not add too much overhead.
An average dequeue rate is created based on the standing queue and used to calculate the current delay. Then, with that obtained value they calculate the packet's dropping probability, so next arriving packets are dropped or not based on that probability. During the course of the connection, PIE will be making adjustments to the probability by keeping track of the delay.

2.5 Conclusion

This chapter has provided a brief overall insight on the inner workings of TCP and it's evolution from it's very beginnings until nowadays. Also, a small explanation enumerating the main characteristics of TCP congestion control has been provided. Finally, in the TCP Performance Extensions subsection all well known TCP extensions that aim to improve performance have been explained and summarized, as they conform a key piece in most of the actual TCP implementations.

Chapter 3

Tools of Trade

3.1 Network Simulators

Network simulators stand as a piece of software that allow users to model a given network by providing a way to create a fictitious scenario able to predict it's behaviour under certain conditions. In other words, it lets users to define a real world networking environment in a single (or distributed) computer by it's pieces: devices, links, channels, etc.. and customize that scenario by allowing them to modify a huge range of parameters that would be taken into account in a real model. The network model takes into accout the pieces of the simulation, such as nodes, links, etc.. and the state transitions or the events of the simulations. A simulator will therefore allow to check and trace the state of every element involved in the simulation (such as packets) by providing an interface to do so, and it will also provision insightful output on the execution of the simulation, in the form of trace files or whatever file format. There exists a really wide range of offer when it comes to network simulation. The available products can be split into two main categories: academic and commercial simulators.

3.1.1 Academic simulators

3.1.1.1 ns3

An open source discrete-event network simulator. It is a complete rewrite from the previous widely know simulator **ns2**. Mainly oriented towards academic research and educational purposes. It is the chosen simulator to perform this thesis' tests. We will dig into it's inner workings in a more detailed fashion more in advance.

3.1.1.2 Direct Code Execution (DCE)

DCE is a framework written in C++ that serves as an extension of the ns3 simulator. It allows ns3 users to execute userspace programs and kernelspace network protocols within the simulator. It is highly configurable and allows the user to write their own protocols and emulate them in the simulator. It is the continuation of the NSC(Network Simulation Cradle) abandoned project.[14]

3.2 Traffic Generation Tools

3.2.1 iperf

iperf is a commandline tool specifically designed for network performance measurement, oriented to bulk data transfer. It works over TCP and UDP, and let's the user tune several TCP/UDP paramaters. It reports bandwith, delay jitter, and datagram loss. It also provides a graphical interface called **Jperf** that can be found here [12]

3.2.2 netperf

netperf is a tool that provides the same capabilities as iperf but also measures some other such as Request/Response or Connection/Close, or even a combination of both. [17].

3.3 Network Measuring Tools

3.3.1 Captcp

Captcp is an open source program for TCP analysis of PCAP files. It is written in python and provides a really simple modular interface to extract information out of a pcap file and plot interesting data. It offers less characteristics than tcptrace.

3.3.2 Tcptrace

It is a well known tool for the analysis of TCP dump files, either PCAP or in other format. tcptrace accepts several packet capture formats and provides a much more comprehensive statistics report than captcp in case thorough analysis is required, such

as elapsed time, bytes and segments sent and recieved, retransmissions, round trip times, window advertisements, throughput, etc..

3.4 Gnuplot

Gnuplot stands as reference commandline tool originally created to allow scientists and students to visualize mathematical functions and data interactively, but has grown to support many non-interactive uses such as web scripting. It is highly customizable and provides a huge range of plotting options.

3.5 Programming Languages

Besides the need of using **C++** as the default language to program the simulations, and excluding the optional **Python** bindings that ns3 provides, during the course of this master thesis there has been a continuous requisite of performing repetitive and trivial tasks, such as the creation of scripts to perform simulations with a varying range of parameters, carrying on automated analysis of pcap files, parsing log and data files, or performing simple calculations in order to get the right data sets for their posterior plotting.
To carry on this work I used the **Bash (Bourne Again Shell)**[7] scripting language. It's simple, intuitive, versatile and extremely powerful to perform practically all system-related needed tasks on a day-to-day basis. In combination of that wonderful tool it has been also utilized the aid of the **Sed (Stream EDitor)**[8] commandline tool and **AWK**[6] text processing programming language. Altogether with the rest of the other default unix commandline out-of-box tools they shape a more than enough working environment to perform all the operations needed.

Chapter 4

NS3/DCE

4.1 NS3

Ns3 is a discrete-event network simulator written entirely in C++ language, with bindings for Python aswell. It is primarily targeted towards educational purposes and research, and it is open source, licensed under the GNU GPLv2 license. As stated on their main website, the main goal on the ns3 project is to develop an open simulation environment for networking research. The simulator is supported primarily for Linux, OS X and FreeBSD platforms.
Ns3 is a continuation of the previous *ns2* simulator, although there is no backwards compatibility between them. The two simulators are both written in C++ but ns3 is a new simulator that does not support the ns2 apis. Some models from ns2 have already been ported from the old to the new one.

4.1.1 Architecture Overview

The simulator follows a modular architecture, in which each module targets and provides different specific functionality as shown in Figure 4.1. It provides models of how packet data networks work and perform, and also a simulation engine to conduct experiments in a highly controlled and reproducible environment, thus allowing users to perform studies that otherwise would be more difficult or even impossible in the real world. It is is designed as a set of libraries that can be combined together with other external tools. Although ns3 gives the user the possibility to use external animators, data analysis and visualization tools, one should expect to perform all the simulator's workflow within the commandline and the regular c++/python developments tools. Really important modules are the **network** and **core** modules, as they will (most likely) participate in

all the simulations, as they comprehend the basic minimum pieces a single simulation must have.
Modules are internally organized in this way:

- model/ : source code for the main part of the module.
- helper/ : code for helper classes (explained afterwards).
- examples/
- tests/
- bindings/ : files related to python
- doc/
- wscript : this file serves as the configuration guideline for the **Waf** builder and provides it with information on how and where to build the module. It can be seen as some sort of Makefile for that tool.

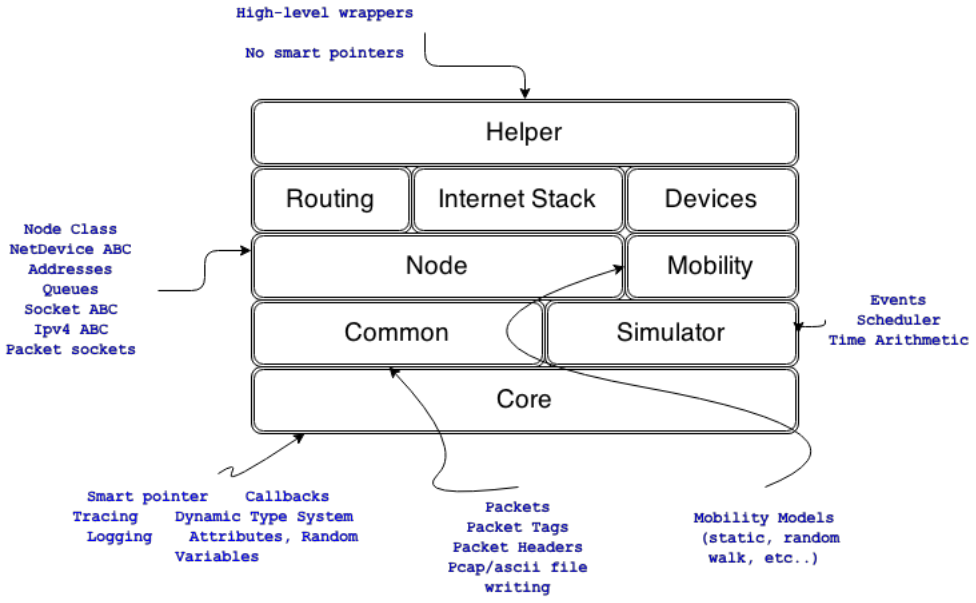

FIGURE 4.1: ns3 architecture

4.1.2 Building NS3

The process of building the ns3 environment can be done using three different methods: by using the **Build.py** script provided inside the simulator's tarball, the **Waf** meta build system, or the **Bake** utility. The first choice is slowly becoming deprecated by the more popular use of the last two tools. It is a simple build script that will configure the project in the most commonly useful way. Several options can be specified.
The *Waf* build system is a python framework for configuring, compiling and installing applications. Among their maybe best features one can mention the ability to specify the building order of the projects and also automatic dependency resolution. Waf reads a configuration file called *wscript* situated in the project's root folder, and will configure and build accordingly to the instructions specified inside. Waf is used for example the **Samba** project, and can be found in [19].
Last one, *Bake*, is an integration tool utilized by software developers to automate the build of various projects which depend on each other and might be developed and hosted by unrelated parties. It was developed to automate the build of the ns3 simulator by taking into account that it can be composed of several independent projects. It works in a simple way: first of all in the configuration phase bake.py will create a personalized configuration file called **bakeconf.xml** with all the modules the user specified he/she wants to be built. Once the file is created, in the download phase the script will only download those pieces the user has stipulated, and in the last phase (build), bake will put all those parts together by compiling and assembling them. Bake.py main page can be found in [2] and the tutorial in [18]

4.1.3 Object Model

The ns3 simulator is in it's true main core a generic C++ object system. What it does mean is that objects can be declared following C++ rules and all the traditional features of those objects and normal operations can be performed. However, ns3 extends the normal object model by adding a new layer of functionality.
There are three special base classes in ns3: *Object*, *ObjectBase* and *SimpleRefCount*. In practice, *Object* is going to be the one that the user will most commonly find. While C++ normal objects provide common object capabilities such as abstraction, encapsulation or inheritance, those objects that inherit from one of the last thee mentioned, will be benefit from the following properties:

- The ns3 type and attribute system
- The Object Aggregation system

- The Smart-Pointer reference sytem (Ptr class)

The aggregation and smart-pointer systems furnishes script programmers with a mechanism that facilitates the use of the memory management and inheritance system, aswell the absence of memory leaks.

4.1.4 Nodes

Network topologies in ns3 are defined using the simple concept of Node. Those are simple abstract elements that one might think at first as simple bare-metal computer shells. Then, by using the aggregation pattern we can add NetDevices, Applications, Stacks, and several components that would also be added in a real network. Net(work) Devices can be viewed as network interface cards, and several Applications can be run on top of those NetDevices using Stacks (which interface with those NetDevices). Just like a real system, NetDevices can work on different stacks such as Ipv4, Ipv6 or any other.

4.1.5 Channels

Ns3 provides models to represent several different network channels to work on. By using the helper api, ns3 allows to create sets of net devices that share a common channel, such as *CSMA*, *Point-To-Point*, *WiMax*, and others. Because of the wide range of potential scenarios and channels that one could use to model real life scenarios (also WiFi, Mesh, ..) this thesis will primarily simulate Ethernet based networks, which can be achieved by the *CSMA* model, explained with a little more detail afterwards, that simulates a simple bus network in the spirit of Ethernet but with some defficiencies.

4.1.6 Helper API

The high-level helper api is created as a solution to aid programmers to avoid writing long and tedious code by using all the time low-level object managing. This API's main goals are:

- Independence with the low level api, being able to code indistinctively.
- Containers: it's most useful feature: containers allow a way to apply a number of identical actions to groups of objects. In a way for example to create a Csma channel with two Csma net devices:

```
1       NodeContainer csmaNodes;
2       csmaNodes.Create (nCsmaNodes);
3       CsmaHelper csma;
4       csma.SetChannelAttribute ("DataRate", StringValue ("100Mbps"))↩
    ;
5       csma.SetDeviceAttribute ("EncapsulationMode", StringValue ("↩
    Dix"));
6
```

4.1.7 CSMA (Carrier Sense Multiple Access) Model

CSMA model consitutes one of the main pillars of network topologies studied in this thesis. It provides a way to represent a bus Ethernet-ish network. Nonetheless, CSMA only fulfills a portion of the process, for example, collisions in the Ethernet fashion never happen. Still, provides very useful functionality. The *CsmaChannel* models the transmission medium. Link parameters such as **DataRate** and **Delay** from the media can be accessed. There is no restriction or limitation on the number of devices allowed to be connected or the DataRate specified. Therefore, it is really important to pay special attention by the time to design and run simulations to adjust to real world scenarios, as the simulator will not warn whether or not the parameters you are establishing fall into impossible ranges or values.
The *CsmaNetDevice* attempts to somehow behave as an Ethernet network device. Useful real world network parameters such as the transmit queue or the receive error model can be attached. Creating a CSMA network is done by using the *CsmaHelper*, and installing previously created node container into it, therefore installing the *CsmaNetDevices* into the nodes and communicating them through a *CsmaChannel*

4.1.8 Packets

The Packet framework in ns3 aims to provide a layer of representation of real packets that allow the interconnection between the simulation environment and a real world case. A Packet in ns3 stores a byte buffer, a set of byte tags, a set of packet tags, and metadata. The byte buffer stores the serialized content of headers and trailers stored in the packet, therefore, the representation of those headers is expected to match a real world packet as faithfully as possible.

4.1.9 Events

Ns3 behaves as a discrete-event simulator. It tracks the amount of events that are scheduled to execute at a specified time, and the simulation time moves in discrete jumps from event to event. Once an event is completed, it will go for the next one. To make it happen, the simulator has access to a queue where events are stored and be executed over time. The method *Simulator::Run()* gets all started, and simulation will end at a specific time sated by *Simulator::Stop()* or when all the events end.
Time in ns3 is stored as a large integer in order to facilitate portability and avoid discrepancies accross platforms. Default time resolution is in nanoseconds, but can be set to other resolutions.

4.1.10 Error Model

It is possible within the simulator environment to attach an error model to one or several NetDevices in order to simulate packet drop. This model, part of the network module, is used to indicate if a packet that arrives into the NetDevice should be dropped accordingly to an empirical or stochastic distribution. Several kinds of error models can be used, as *RateErrorModel* or *BurstErrorModel*.

4.1.11 Logging

Ns3 provides a whole framework for enable logging capabilities for the user. It can be specified by program statements or by using environment variables. Logging is mainly for debug builds of simulations. Those logging statementes are used to log different program execution events that occur according to an log event hierarchy. An event happens and can be logged or not if it's event type has been set to be logged.

4.1.12 Tracing

The main purpose of running a simulation is to generate an output to perform further analysis. Tracing subsystem allows a programmer to do so. It is a mechanism designed to allow the user to follow the behaviour and transformation of those items of interest during the simulation, and only getting the information needed.
Tracing system works by using the Callback mechanism, and divides it's way of working between what they call *Trace Sources* and *Trace Sinks*. In general lines trace sources can be seen as generators of events, and trace sinks as as consumers of those events. TCP congestion window for example is a normally used candidate to trace during the

execution of a program.
However, channel helpers provide users with the ability of generating output from a whole simulation into a ascii file or a pcap file, therefore allowing to utilise network analysis tools to extract information from the simulation, from wireshark to captcp. This mechanism to analyze traffic parameters will be the one used in this thesis. In addition, DCE provides provides the *SysCtl* interface (explained after), that allows to interact directly with the TCP stack throught sysctl commands, and thus giving access to tcp parameters in real time.

4.1.13 Randomness

During the course of executing simulations, a need for providing statistical information about the reliability of the measures taken arises. To achieve a significant degree of trust in the experiments made, many independent executions from a single script must be run and results of all of them ought to be correlated following any kind of statistichal measure to ensure the obtained data is trustworthy. In other words, there is a need for providing some level of randomness to the simulations in order to get different results in each single or group of executions.
The way on how ns3 provides randomization is by the use of a built-in pseudo-random number generator (PRNG). This generator provides a long sequence of numbers. That sequence can be partitioned into disjoint *streams*. Each stream is a subset of that big sequence of numbers constituted by numbers chosen following a specific random variable distribution, such as *UniformRandomVariable*, *NormalRandomVariable*, or many others that the simulator provides. Conceptually, the idea to achieve the maximum level or randomization is to hope that two substreams of the random number sequence are uncorrelated.
When a simulation starts, the PRNG uses a seed value to create the number sequence, if that value is the same, each execution will have the same results. One way to modify this is by changing the seed number at the beginning of the execution. However, mutating the seed does not guarantee that two substreams of random variable objects from that sequence will not overlap. Therefore, the way to ensure independence is by changing another parameter called the run number. This number represents the number of the substream of a random variable inside the sequence. In that way you will ensure that you get an uncorrelated stream.

4.2 DCE

4.2.1 Direct Code Execution

DCE is a module for the ns3 simulator that allows to execute nearly unmodified native applications and Linux Kernel code int he context of the ns3 simulator. It is a framework that provides the ability to run existing implementations of userspace and kernelspace network protocols or applications without source code changes. For example, users could run the existing native **iperf** traffic generation tool instead of programming an application inside the simulator that performs the same operations as **iperf** does.

4.2.2 Features

The framework provides a wide range of features:

- Almost full POSIX socket support
- C or C++ native applications
- Interaction with the linux kernel via sysctl interface
- Easy debugging with valgrind and gdb
- Variance of network stacks to work with: ns3 implementation, NSC(network simulation cradle), or native linux kernel tcp stack.

4.2.3 The SysCtl Interface

The *LinuxStackHelper* class inside the DC api bestows two methods: *SysCtlGet* and *SysCtlSet*, that providing the user has installed the Linux Kernel Stack in the node's before by using DCE capabilities, it serves as an interface to execute sysctl commands into the emulated kernel inside the simulation at runtime, therefore giving the ability to change parameters as the congestion window or adjusting send/receive buffers among others. It is fully supported for the Linux Kernel versions that work inside DCE, but is not present in the FreeBSD stack.

4.2.4 Architecture

This framework's architecture is divided intro three layers: **virtualization**, **kernel**, and **posix** layer, as seen in Figure 4.2

First one handles the virtualization of stacks, heaps and global memory. DCE executes all simulated processes inside a single host process, making possible to synchronize and schedule each one without recurring to synchronization mechanisms, and debugging simulations in a simple manner without the help of distributed debuggers. Kernel layer embeds network protocol implementations found in the Linux Kernel and makes them communicate with ns3. At the top of the network stack, application level data is exchanged using socket-based applications using the kernel socket data structures. That allows ns3 to access kernel structures via the sysctl interface for example. Last, the posix layer is a try to be the replacement for the traditional **glibc**, by implementing first of all the subset of features that most common native applications need, and then incrementing the amount of apis covered. Still, DCE doest not provide full api coverage.

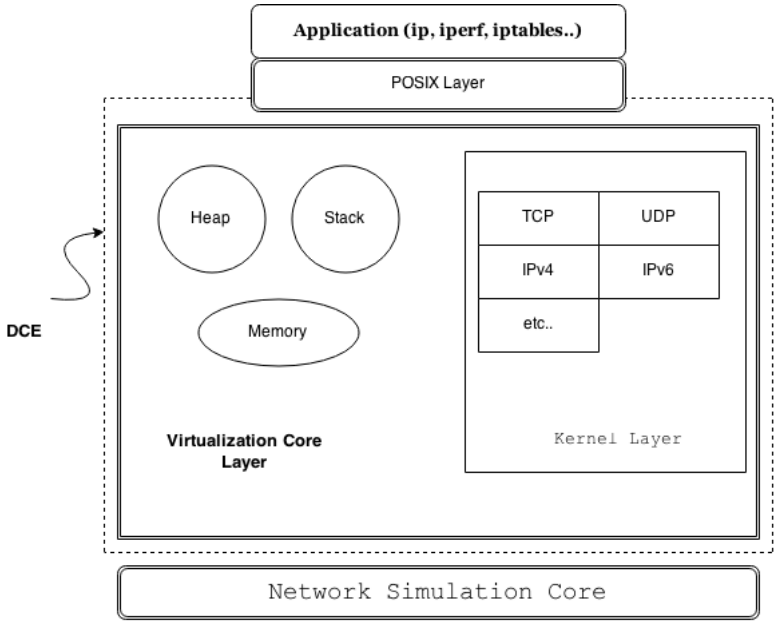

FIGURE 4.2: dce architecture

4.2.5 Extending DCE

There are two ways on how you can extend DCE's capabilities: by using your own userspace protocol implementation (i.e. a native executable not tested yet), or by using your own kernelspace protocol implementation, and making the current kernel version in DCE support it.

For the first case, DCE needs to relocate executables in memory, so the application must be recompiled with flags that establish the binary is PIC (Position Independent Code).

For the second case, you have to reconfigure and recompile the kernel, and optionally if needed, adding new system calls to DCE api.

4.3 Measuring Concepts

The two main measures that will be observed during the course of experiments are execution *delay* time (without taking into account simulator time) and *throughput* (goodput, explained afterwards). **Throughput** is the rate at which data is traversing a link, or with another words, the volume of information that flows accross a network. We will focus in a derived measure from this one called **Goodput**. Throughput itself is the measurement of all data going through a link, whereas Goodput focuses on *useful* data only. Troughput measurements can include undesirable data such as segment retransmissions or overhead data, being protocol wrappers an example, thus distorting the spectrum of our analysis.
TCP connections can be divided in *flows*. One flow identifies traffic going from the sender to the receiver of the connection, and the other flow is the "coming flow" from the receiver to the sender, mostly packets acknowledging segments being received by the other end. In order to get the most accurate measurements the flow going from the sender towards the receiver is the one taken into account as it is the one who carries the actual transmission of **Application Layer Data (Payload)**. Connection time (**Delay**) can be measured by substracting the timestamp value of the *FIN* packet and the first **SYN**, both values extracted from the pcap file. The elapsed time in between is the real time of a TCP connection.

4.4 Simulation Environment & Workflow

The way experiments are performed in this thesis follow a well defined workflow. The combination of NS3 simulator/DCE framework allows to represent a wide range of heterogeneous network topologies inside the simulation environment and to emulate Linux Kernel Stack, allowing to using native applications too, hence giving a real taste of reproducible real world experiments.
Simulations are executed in each of the kernel versions supported by the stable git branch of DCE. At the moment of writing this thesis, available stable versions are: *2.6.36*, *3.7.0*, *3.10.0*, *3.12.0* and *3.14.0*.
For each kernel version, as a base, simulations will be run for every tcp congestion control algorithm avaiable in all those versions. Common accesible agorithms are: **reno**, **bic**, **cubic**, **westwood**, **highspeed**, **hybla**, **htcp**, **vegas**, **veno**, **scalable**, **lp**, **yeah**,

illinois.

There are significant changes in the TCP stack for each version, so this group encompasses a sufficiently heterogeneous set to provide variability accross tests just by switching kernels. However, the scope of experiments will be limited to the first four versions, as at the time *3.14.0* has shown inability to change TCP stack parameters via the SysCtl interface (i.e. congestion control algorithm), thus being consequently excluded.

Tracing mechanism is not used, but instead, simulation's output is written to files in pcap format, easing ulterior offline processing, in order to extract statistic data and taking several measures. To do so, as described in chapter 3, *Captcp* and *Tcptrace* tools will be used. The former has been proven to be a good addition to the well known low level packet analysis tools such as tcpdump or wireshark, and statistical results and graphs that generates agree with the ones generated via the latter, proven to be robust and widely accepted.

Once the desired data is extracted from the pcap files it is arranged, ordered and classified in a format that is able to be plotted with the GNU gnuplot plotting tool.

When a single or a group of simulations are finished and data is extracted and plotted, or represented in the most suitable fashion to be checked, it is then analyzed. Upon this analysis conclusions are extracted and may arouse the need or the interesting possibility to change one or several parameters of the tcp stack (either in a single kernel version, or all of them), or some external ones related to the topology and simulation itself and then rerun the tests to see how those new made changes differ from the previously measured data.

Chapter 5

Simulations

5.1 Simulation 1

5.1.1 Topology

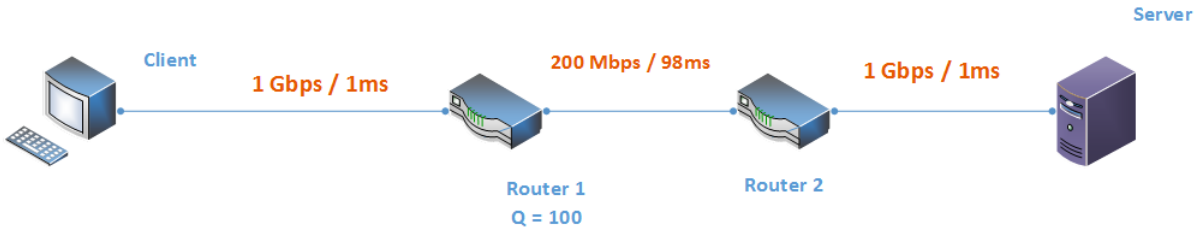

FIGURE 5.1: Simple Topology Simulation

5.1.2 Motivation

This kind of topology was chosen inspired in the simplest case of a dummy network presented in a variety of TCP performance comparison papers: a client communicates with a server sepparated by two routers in the middle by using a single flow and the network between routers is low bandwidth/high latency so there is a drop of throughput and packets are enqueued even when there is a single flow.

As seen for example in [1], [20], or [16]. Most of the papers start from two competing flows in order to generate congestion from the beginning. We will run this very simple scenario at first with no competing flows and different congestion control algorithms to see how do they perform in an initial basis.

There is one important note: there is no agreement on the best congestion control paradigm for high BDP (Bandwidth-Delay Product) networks. An algorithm, whose

performance are optimal in a particular scenario, may perform unsatisfactorily in other scenarios. Moreover, different testbeds lead to different results due to minimal differences in the algorithm implementation or in the network scenario design. Taking execution delay time and throughput as measures serve as a strong starting point.

5.1.3 Description

As seen in 5.1, this scenario consists of four nodes trying to emulate a highspeed network with a congested link. Below here is the simulation's characteristics and specifications.

- Iperf as the traffic generation tool from the client to the server.
- The client will send a fixed amount of bytes to the server: *5 Mbytes*.
- Both routers **R1** and **R2** have two send queues attached with a capacity of *100* packets.
- Queuing discipline taken for both queues is the simple DropTail scheme.
- The lateral networks composed of the **(Client − R1)** and **(R2 − Server)** simulate an Ethernet-like link with the following parameters:

  DataRate 1Gbps

  Delay 1ms

- Central network composed of **(R1 − R2)** is the bottleneck link, its parameters:

  DataRate 200Mbps

  Delay 98ms

- Simulations will be run using packet loss rates of *0%*, *2%*, *4%* and *5%*. Error model to simulate that packet loss is attached to first router (R1).
- For each execution the **Random Number Generator (RNG)** seed is initialized to a random value between 1-100. Seed is initialized with the *time()* C++ standard api function. Run number will be assigned randomly using a different seed each time too.
- **MSS** = 1460 bytes.

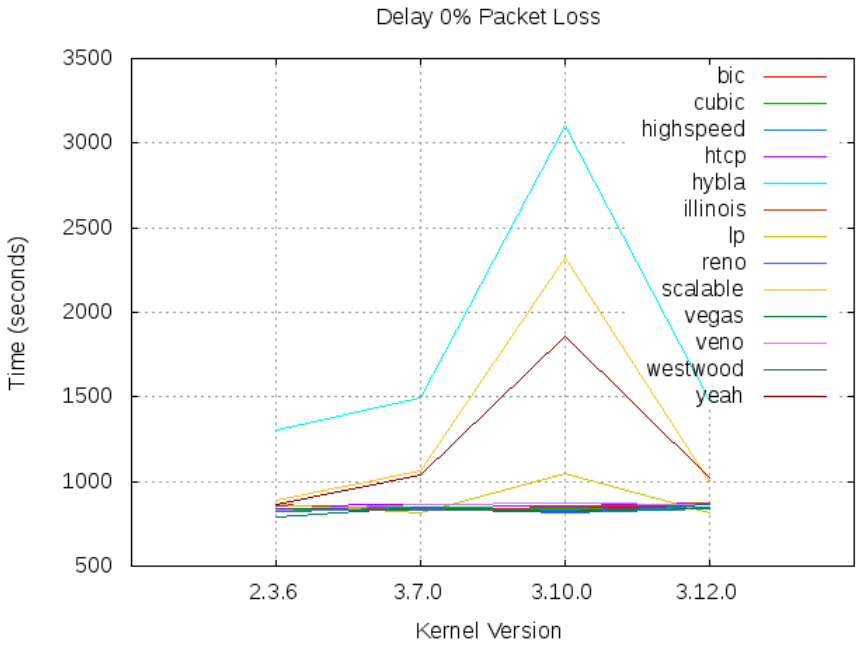

FIGURE 5.2: Delay 0% Packet Loss

5.1.4 Results Without Packet Loss

5.1.4.1 Delay

Execution delays are shown in figure 5.2. Although it is not the way to show results because there is no continuity from one kernel version to the other, seeing the initial results in a colorful way helps to see the superficial differences between algorithms and kernel versions.

As explained in Measuring Concepts, only connection delay time will be measured, excluding simulator time. All of the congestion control algorithms share almost the same connection times for kernel version *2.6.36* (between 600-800s) except from Hybla which takes up to 1700 seconds, approximately 1000 seconds more than the other algorithms. In this scenario, except from Hybla, Lp, Yeah and Scalable the other algorithms share really close execution times, this is due to the lack of proper network congestion (There is only one TCP flow that sends data). There is absolutely not a clear winner. Veno, Westwood and Vegas share practically equal execution times in four kernel versions, so there is no clear conclusion on either which pair of kernel version / congestion control algorithm performs the fastest due to the lack of big differences in this context.

It is however interesting to take a look at the other four algorithms that stood out. Three of them (yeah, scalable, hybla) experience a proportional increase of 400 seconds in version *3.7.0* (all of them where increased by the same factor). Nevertheless, when performing several tests, lp and hybla tend to overlap, sometimes being the former the

Algorithm	Kernel	Avg. Throughput (Kb/s)
bic	2.6.36	7.152
cubic	3.12.0	7.238
highspeed	3.12.0	7.190
htcp	2.6.36	7.233
hybla	2.6.36	5.993
illinois	3.10.0	7.234
lp	3.7.0	6.527
reno	3.12.0	7.346
scalable	3.7.0	7.413
vegas	2.6.36	7.255
veno	3.10.0	7.266
westwood	3.12.0	7.131
yeah	3.10.0	7.288

TABLE 5.1: Congestion control algorithms, kernel versions, Average throughput 12 iterations

one with the highest time and some others the latter, but always moving in the 2000-2500 seconds interval. Yeah on the contrary may jump from 1500 to 2700 seconds depending on the simulation.
In version *3.10.0*, lp sees a small growth in execution time, but then again the three outstanding algorithms see huge spikes in their delays. In next version (*3.12.0*) their execution times drop to the same times they were obtaining in previous version.

5.1.4.2 Throughput

If we plot the throughput graphics for all congestion control algorithms the resulting graphic is illegible. Instead, we will be guided by table 5.1. Each row represents the highest average throughput value for each algorithm between the four kernel versions where it was executed. Average throughput is calculated by obtaining the mean value over 12 iterations. In this scenario most of the algorithms and kernels share a very similar throughput except *hybla/2.6.36*, being the best one in this case *scalable/3.7.0* by almost no difference with the others. Values have been rounded from 14 decimal digits to 3.

In order to know if throughput average values for scalable can be trusted we can serve from the errorbars plot in 5.3. This graphs represent the mean value of twelve executions of the simulation for each kernel version, and the standard deviation (stddev from now) from that given value.

We can see that although the most trustworthy measure would remain in kernel *2.6.36*, the one with the lowest stddev, the one seen in *3.7.0* is low enough to be considered

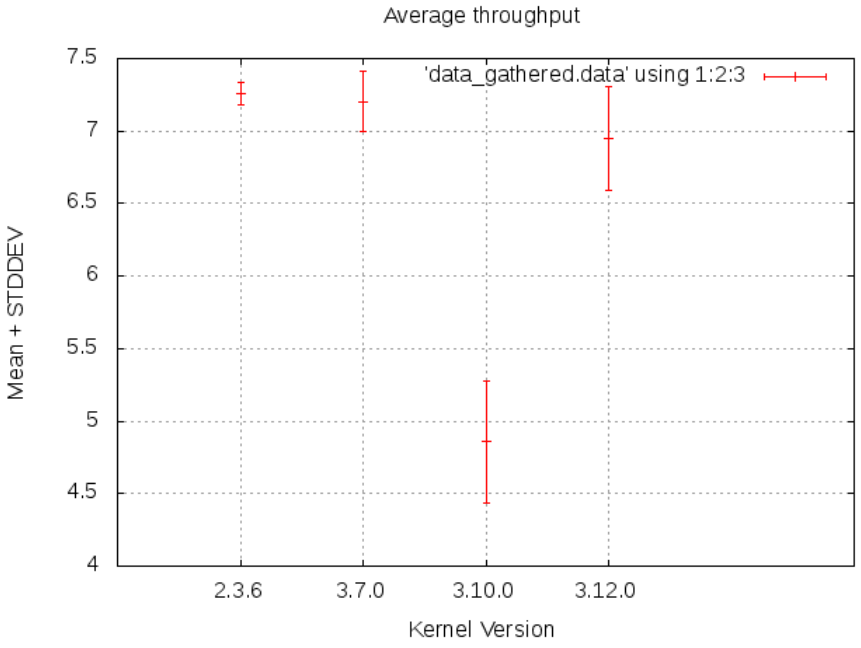

FIGURE 5.3: Throughput 0% Packet Loss ScalableTCP

as valid. As one may remark in this case the average throughput from twelve executions is higher in *2.6.36*, but very closer to the winner in the first place. It would change slightly among executions being sometimes one sometimes the other. Remaining kernel measurements are not taken into account, as seen *3.10.0* provides a very low mean throughput value, aswell as *3.12.0*, and also in earlier measurements the average throughput measured is substantially lower than their first two versions. It is therefore acceptable to assume that *3.7.0* was the winner in this case, with a mean value close to version *2.6.36*. Out of curiosity we may check the average mean values of all congestion control algorithms grouped by those winner kernel versions: 5.4 and 5.5.

Indeed, for version *2.6.36* scalable has one of the highest average throughputs together with htcp, cubic and veno. The worst ones are by far hybla and lp. Hybla not only has the lowest throughput but also the highest stddev value, which means that it has much more packet losses. In *3.7.0* we can see that most of the algorithms share throughput values between 7-7.4 Kb/s except from hybla, again performing extremly poorly like we could see with version *2.6.36* in figure 5.4. Also reno and westwood's stddev values are more away from the mean than the other algorithms, proving they provoke more packet losses than other algorithms in this version.

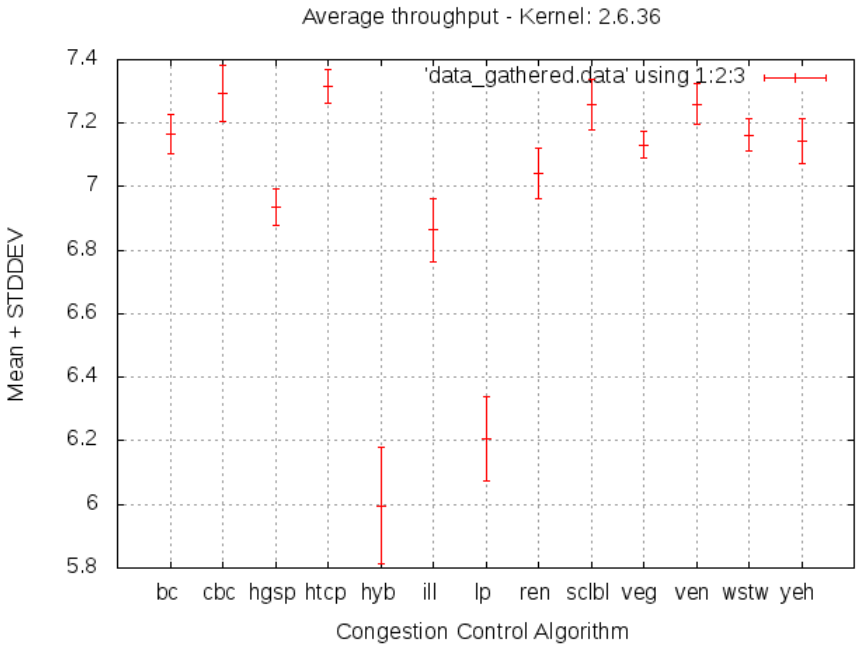

FIGURE 5.4: Throughput 0% Packet Loss kernel 2.6.36

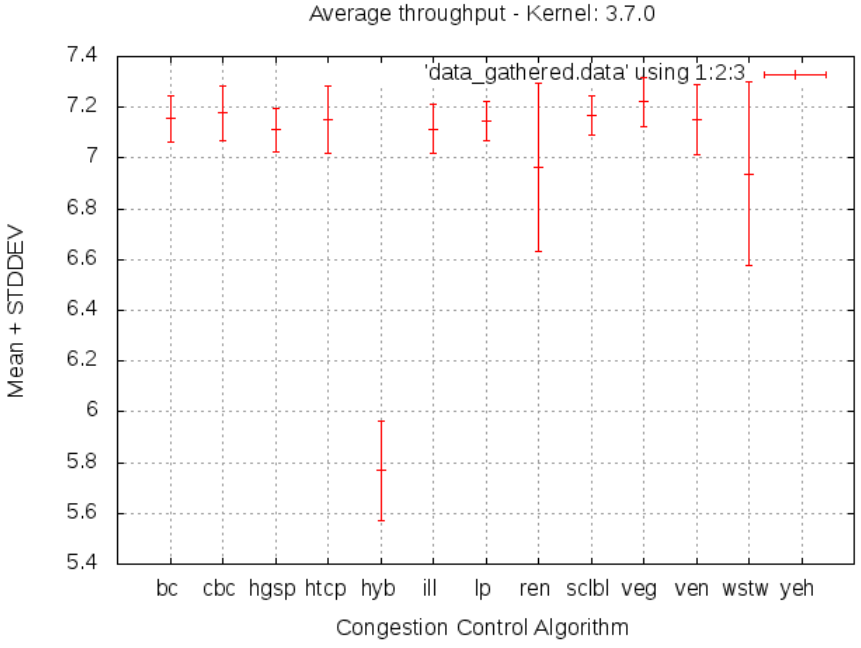

FIGURE 5.5: Throughput 0% Packet Loss kernel 3.7.0

5.1.5 Results 2%, 4% and 5% Packet Loss

Results obtained by performing simulations varying the packet loss parameter are negligible, as expected. Graphics for either throughput or delay measures are practically equal due to the lack of network congestion. In a real environment, packet corruptions and losses associated with physical media are much lower than **1%**, being the big majority caused when several flows compete for limited network resources.

5.1.6 Observations

Observations will be done taking the data and graphics from the 0% packet loss case, as there is almost no difference with the other versions, being this ones perfectly applicable to the other secenarios.
As seen in 5.1.4, there are four congestion control algorithms that stand out excessively due to their slowness: hybla, scalable, yeah and lp. They experience delay growth accross every kernel version, and suffer a huge spike concretely in *3.10.0*. First of all, regardless of the kernel version being executed, it is necessary to understand the reasons on why those algorithms are not performing properly in this very specific scenario.
Hybla TCP: this algorithm was designed with a single goal in mind: to deal with the TCP satellite problem [3] or in other words, networks that experience very long RTT's. As an strategy, it takes a fixed value of RTT and tries to scale TCP Reno in order to behave as a Reno connection with an RTT of RTT_0. The formula that explains cwnd changes expressed in terms of current time ($t0$) is

$$cwnd = (t - t_0)/RTT + cwnd_{min}$$

Hybla uses a factor to increase the cwnd instead of normal value of 1:

$$\varrho = RTT/RTT_0$$

so cwnd grows in the way:

$$cwnd = cwnd + \varrho^2$$

which would mean that in a high RTT network, the ϱ quotient would be much higher that 1 and cwnd would grow faster, causing a dramatic increase of packet drops in the receiver.

By taking a look at the RTT graph of hybla in worst delay scenario (*3.10.0*)(see graph 5.6) there are very big spikes followed by sudden drops, following quite a stable interval, which would coincide with each windowful delivery of traffic. Comparing this RTT

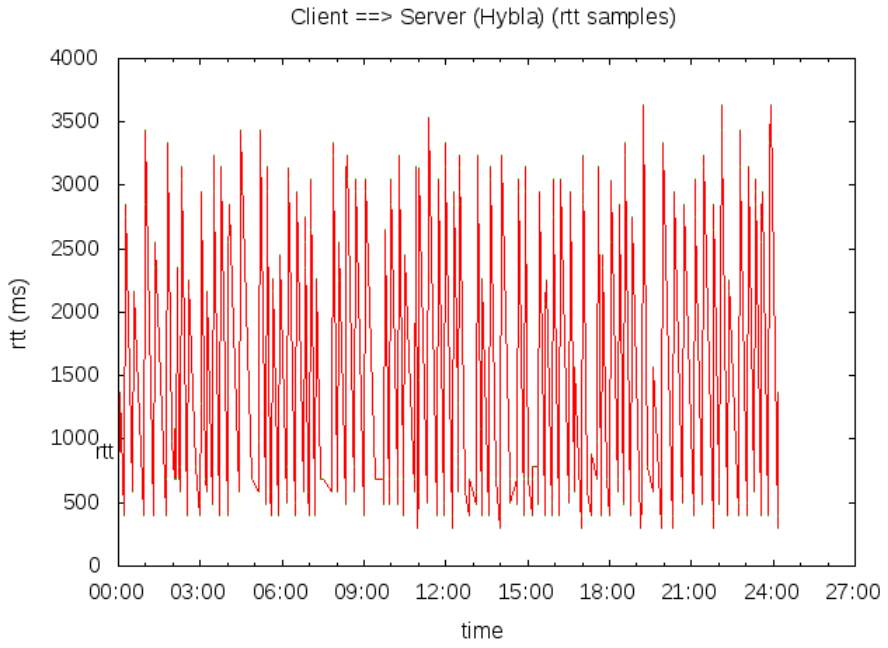

FIGURE 5.6: RTT Hybla kernel 3.10.0

graph with one of the "winning" algorithms such as veno (see graph 5.7) one can see the RTT is varying but in a much narrow interval during the connection, because veno follows a not-so-aggressive cwnd increase rule and packet drops are not that common.

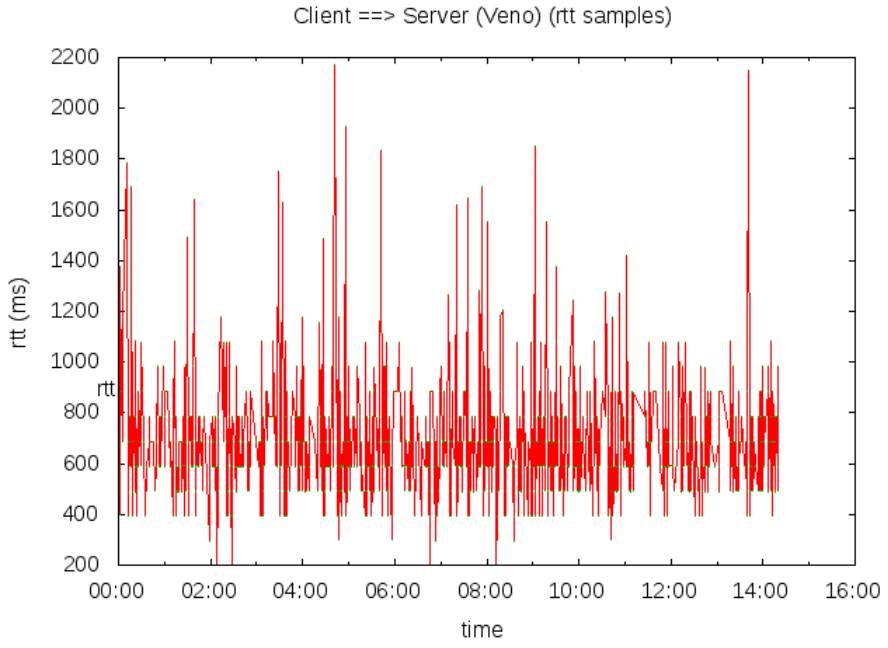

FIGURE 5.7: RTT Veno kernel 3.10.0

Compared to other congestion control algorithms, hybla does not estimate the RTT by

looking at packets arriving, but takes the measured RTT and divides it to the arbitrary RTT_0 in order to get the scaling cwnd factor. Hybla is supposed to work well in high RTT environments, where by using the RTT/RTT_0 factor it can force the sender to increase congestion window really quickly and therefore sending a bigger amount of packets in one RTT. It is highly suggested to have $SACK$ enabled in the receiver side, in order to allow faster recovery from multiple packet losses, although it makes no sense running a simulation on hybla with SACK disabled either ways also all the other congestion control algorithms would have their performances impaired.
TCP Timestamps is an option also strongly recommended when using hybla, as the actual RTT measurement can be estimated with more precision. The receiver echoes back the timestamp in the corresponding ACK, thus allowing more accurate measurement by the receiver of the actual RTT.
Scalable TCP ([13]) shows one of the slowest execution times in this simulation. It works by providing server side alteration of the TCP windows update algorithm. It aims to improve bulk transfer in highspeed wide area networks, when tcp connections use large window sizes. The congestion window growing rule is

$$cwnd = cwnd + 0.01$$

whereas when packet drop is detected, it cut's the window size in the way

$$cwnd = cwnd - [0.125 * cwnd]$$

so this algorithm makes the congestion window grow slower than other ones. However, it also makes it decrease slower, avoiding cutting into half with the general AIMD (Additive Increase, Multiplicative Decrease general technique in congestion control:

$$cwnd = cwnd/2$$

If congestion window is not large, in a network when there is a slow link at any point (as the one in this scenario), it would be all the time full and thus involving the connection to deal with more packet drops.
YeAH TCP is a delay-aware state-enabled congestion control algorithm. Through delay measures, when the network is sensed unloaded it will quickly exploit the available capacity, trying to keep the network buffer utilization always lower than a threshold. Moreover it is designed to be internal and RTT fair, Reno-friendly and resilient on lossy links. Yeah works in two main modes: "Fast" and "Slow". In the Fast state cwnd is increased aggressively, and in Slow one it acts like TCP Reno. It decides its state and acts consequently. The state is decided according to the estimated number of packets in the bottleneck queue. Two parameters are measured RTT_{base}: the minimum RTT

measured by the sender, and RTT_{min}: the minimum RTT estimated in the current cwnd. From those two values another one is extracted as its substraction $RTT_{queue} = RTT_{min} - RTT_{base}$. This new value serves to estimate the number of packets enqueued:

$$Q = RTT_{queue} * G$$

being G the goodput. Also, a parameter to estimate network congestion is calculated out of previous RTT's:

$$L = RTT_{queue}/RTT_{base}$$

From Q and L the state is decided by taking the condition: if $Q < Q_{max}$ and $L < 1/\varphi$ then algorithm is in "Fast" state, otherwise in "Slow"; being Q_{max} and φ tunable parameters in the network. This alternation between "Fast" and "Slow" state in the algorithm can be verified by checking it's inflight data segment's graph in the worst performing kernel, *3.10.0* (see 5.8) and comparing it with the inflight data graph of TCP Reno, the way it behaves in "Slow" mode (see 5.9)

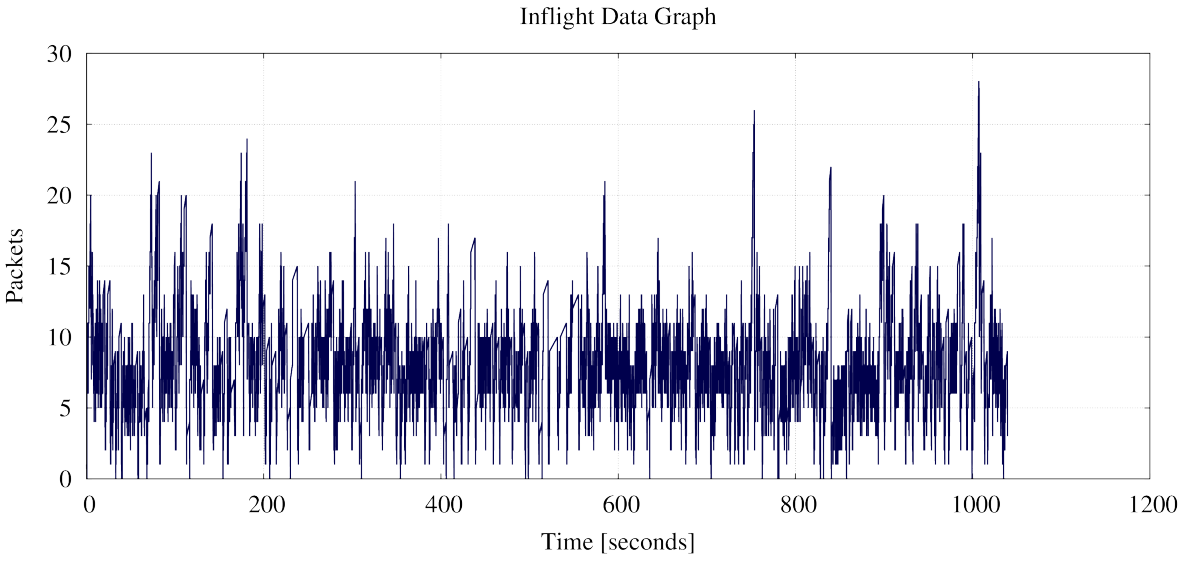

FIGURE 5.8: Inflight segments YeAH kernel 3.10.0

Reno maintains an approximate $cwnd = 6$ all along it's execution time, whereas YeAH jumps from bursts of $cwnd = 20$ to $cwnd = 10$. It means that due to this scenario's characteristics, YeAH alternates between aggressive sending bursts of packets, (the high

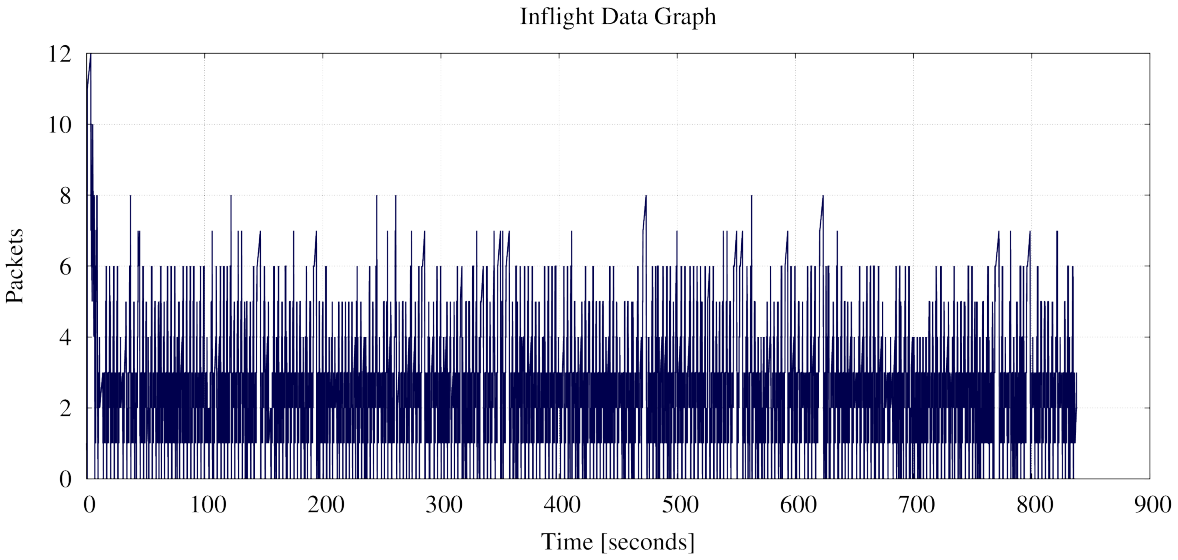

FIGURE 5.9: Inflight segments Reno kernel 3.10.0

spikes in the graph) with the "Slow" mode, where it matches with the Reno graph. This doest not guarantee that this behaviour is caused by congestion because Reno could be simply underutilizing the network link, however, analyzing the pcap trace of both algorithm's execution we verify that

- Reno = 445 packet retransmissions, 10.93% per packet.
- YeaH = 1611 packet retransmissions, 30,77% per packet.

Which indeed shows that YeAH has a packet drop rate around 20% higher than Reno's. The reason behind this is that while Reno is maintaining a conservative approach in this simple high-BDP network, YeAH increases it's sending rate aggressively overflowing the first router's queue and therefore involving many more packet drops & retransmissions.

lp TCP: it was thought as an approach to develop an algorithm that uses excess bandwidth for TCP flows. It can be used for low priority data transfers without "disturbing" other TCP transmissions (which probably don't use TCP Low Priority). The disturbing fact is that it is fairly affected in version *3.10.0* but in the rest of kernels it's execution time is alike the plurality of algorithms.

Many times during this section there have been references to **TCP Reno**. This was one of the first tcp congestion algorithms, continuation of **TCP Tahoe** that neither adds nor modifies any rule to manipulate the cwnd, but it is the first algorithm that implements Slow Start, Fast Retransmit and Fast Recovery.
Now, after gaze is pushed aside the **Why's** and **How's** related to the algorithms, one question may come into mind:
Why does kernel version 3.10.0 performs so badly and executes so slowly?. Taking a look at that version's changeset we see among the prominent features the implementation of Tail-Loss Probes (TLP) and the Forward RTO (F-RTO) mechanism, aswell as removing the TCP Cookie Transactions ([RFC6013]).
Checking the differences between TCP parameters by looking inside `/proc/sys/net/ipv4`, we can see that the major differences with previous version *3.7.0* are:

- `tcp_early_retrans=3`
- `tcp_max_orphans=16384`
- `tcp_max_syn_backlog=128`
- `tcp_max_tw_buckets=16384`
- `tcp_mem=90471\\120628\\180942`

The first one indicates that Early Retransmit and TLP are activated, and the last one does not differ too much from the values in *3.7.0*. `tcp_max_orphans` regulates the maximum allowed quantity of orphaned (not attached to a file handle) connections. It is a security issue and therefore not strictly bound to the lack of performance.
`tcp_max_syn_backlog` represents the maximum number of queued connection requests which have still not received an acknowledgement from the connecting client. In this case not important because there is only one single connection.
`tcp_max_tw_buckets` are the maximum number of sockets in *TIME_WAIT* state allowed in the system. Again there is only one single connection (socket) and does not affect.
Data results won't change heavily accross scenarios (0%-5%) because it is a simple topology with only one flow and even though having a slow link that creates congestion, there are no competing flows for network bandwidth that will test the algorithm's capabilities when it comes to balance equal fairness and link usage amongst them. That is why with different packet losses most of the new results won't be shown, because they are practically equal to this case's.

5.2 Simulation 2

5.2.1 Topology

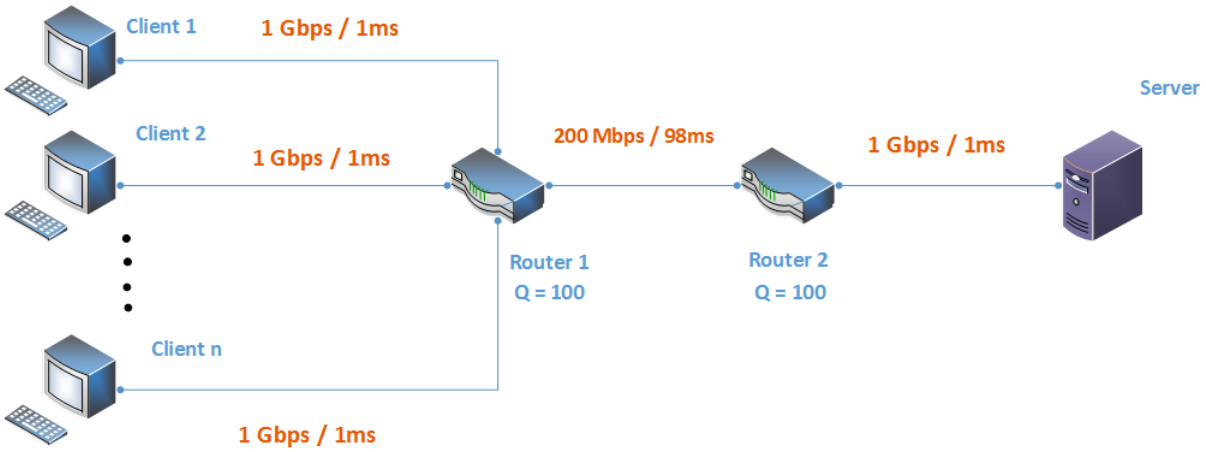

FIGURE 5.10: Multiple Flow Topology Simulation

5.2.2 Motivation

As it is shown in above's topology: 5.10, this simulation shows an approach much closer to a real world example, when several flows compete for the bandwidth and network resources, sending an receiving data.

The amount of congestion control algorithms taken into account in previous simulation is excessive. Measuring too many variants unnecessarily increases difficulty because many of them are just redundant versions of some others that are already present, or they are old and obsolete.

Several ways of classifying and dividing the algorithms is possible in function of some common features: design-oriented (wireless, wired, satellite ..), network congestion detection, the way they tangle with the congestion window to increase/decrease it, etc.. A good taxonomy of the algorithms is by taking into account the way they have to detect network congestion: loss-based, delay-based and loss-delay based. First group considers packet drop as an implicit declaration on network congestion, second group considers packet delays instead, and last one uses a mixture of both. It is therefore useful to reduce the total amount to a subgroup that encompasses representative examples of each type. Several algorithms from the initial set are variations and refinements of others, some are deeply deprecated and some others were designed to perform exceedingly in a specific environment other than the one simulated in this scenario. Cubic is a modification of bic so the latter will be discarded from the group.

Now let's pay attention to this four algorithms: reno, veno, westwood and vegas. All

of them share a loss-based strategy. Westwood stands as a sender-side modification of NewReno which is in turn another modification of reno. Veno constitutes a combination of both vegas & reno algorithms and thus it incorporated both algorithm's capabilities. Veno is oriented to wireless scenarios, however in a wired link veno performs practically equal to reno with a slight performance increase. Certain tests in this multiple flow scenario before this filtering showed that veno performs a little better than westwood. Hence, veno is chosen to be measured and the rest are discarded.
Lp follows a delay-based strategy focusing in taking advantage of all available bandwidth when it is being unutilized, which constitutes and interesting behaviour to study with respect to other algorithms that take network resources aggressively.
YeAH is another variant worth being taken into accout. It follows a loss-delay mixed approach and it is compatible with other reno flows. It's surprisingly low performance in 5.1 makes it interesting to be added to the measurement group.
Highspeed TCP's RFC [RFC3649] dates from *2003* which makes it pretty obsolete. A quick look into it's description states that performs poorly in high-BDP networks. Both previous statements make it a bad candidate and it is therefore discarded.
H-TCP's draft ([drafthtcp]) dates from *2007*, it follows a loss-based approach and was created with the idea in mind to follow a more aggressive strategy for high-BDP links and to be TCP friendly in low-BDP flows. It is chosen for new measurements.
Hybla is definitely left out of the scope of measurements because it is very purpose-specific, oriented to satellite networks, showed terrible performance and is widely deprecated and obsolete.
Between both remaining algorithms: scalable and illinois, the latter is chosen to present better delay execution times in 5.1 even when they both showed alike throughputs. Illinois is loss-based and is focused on high speed and long distance networks, improving average standard TCP throughput. Initial group was constituted of 13 algorithms. After the "clusterization" 6 algorithms are left, representing each one a set or family of shared features among them: **cubic**, **veno**, **yeah**, **lp**, **htcp**, **illinois**.

5.2.3 Description

This topology : 5.10 follows a structure closer to the first simulation but in this case the left side of the simulation, the "client side" has a variable number of nodes. These are the specific characteristics of this simulation:

- **Iperf** as the traffic generation tool from the client to the server (bulk data).
- Four different sets of nodes sending traffic from the client side: **2**, **4**, **6** and **8** nodes.

- The first node on the client side is the main flow, measures will be taken in this node's network interface, and it will be the only one where the tcp congestion control algorithm is changed. It will send an amount of *1024 Kilobytes*.
- Rest of the nodes in client side will conform the background traffic, sending each one an amount of *4096 Kilobytes* to create and simulate congestion.
- Both routers **R1** and **R2** have two send queues attached with a capacity of *100* packets.
- Queuing discipline taken for both queues is the simple DropTail scheme.
- The lateral networks composed of the **(Clients − R1)** and **(R2 − Server)** simulate an Ethernet-like link with the following parameters:

  DataRate 1Gbps

  Delay 1ms

- Central network composed of **(R1 − R2)** is the slow link to create congestion, its parameters:

  DataRate 200Mbps

  Delay 98ms

- In a physical link normally there should not be any packet losses that exceed a rate of 5%, and that is incredibly high either. In this case congestion is created by the connection of competing flows that fight for the available bandwidth, therefore packet loss will be absent in this simulations, as there will be enough traffic to create losses.
- For each execution the **Random Number Generator (RNG)** seed is initialized to a random value between 1-100. Seed is initialized with the *time()* C++ standard api function. Run number will be assigned in sequentially in order to ensure independent simulations.
- Error bar plots for mean and standard deviation are calculated over sets of 6 runs.
- **MSS** = 1460 bytes.

5.2.4 Results: 2 Nodes

5.2.4.1 Delays

In table 5.2 we can see how for each congestion control algorithm and each kernel version, it shows the average delay time calculated over five executions and it's confidence interval.

Algorithm	Kernel Version	Avg. Delay(seconds)	Conf. Interval (90%)	Interval Range
cubic	2.6.36	115	±32.62	83.26 to 148.49
	3.7.0	230	±23.14	246.42 to 292.71
	3.10.0	205	±85.91	124.27 to 296.1
	3.12.0	395	±16.47	399.23 to 432.17
htcp	2.6.36	166	±48.1	118.3 to 214.51
	3.7.0	269	±23.14	246.42 to 292.71
	3.10.0	210	±85.91	124.27 to 296.1
	3.12.0	415	±16.47	399.23 to 432.17
illinois	2.6.36	257	±14.68	242.86 to 272.21
	3.7.0	291	±17.37	273.83 to 308.57
	3.10.0	232	±70.23	161.95 to 302.41
	3.12.0	312	±20.9	291.51 to 333.31
lp	2.6.36	352	±166.02	186.14 to 518.19
	3.7.0	500	±118.98	381.37 to 619.33
	3.10.0	542	±125.04	417.22 to 667.29
	3.12.0	422	±128.95	293.95 to 551.84
veno	2.6.36	284	±20.39	263.7 to 304.48
	3.7.0	316	±23.87	292.32 to 340.06
	3.10.0	256	±78.85	177.44 to 335.14
	3.12.0	356	±34.21	321.85 to 390.27
yeah	2.6.36	245	±31.92	300.92 to 364.76
	3.7.0	259	±7.88	251.53 to 267.29
	3.10.0	224	±67.43	156.99 to 291.85
	3.12.0	352	±23.04	329.13 to 375.21

TABLE 5.2: Cong. control algorithms, Avg delays and statistics, 2 nodes

This table's target is avoiding plotting all the error bar plot graphics for each algorithm sepparately. Conf. Interval represents the Confidence Interval with a confidence value of **90%**.

Cubic stands as the fastest algorithm of all versions but *3.12.0*, which turns to be illinois. Htcp also casts time execution delays closer to cubic's in this low traffic scenario. Delays are only sensibly lower in cubic compared to htcp though, therefore one can deduce it is due to the lack of high congestion that differences are not yet that pronounced. Still, in average cubic shows the smallest time. By checking errorbar plots for the winner (cubic, 5.11) one can graphically see how for the winning version cubic shows the lowest standard

deviation. This means that all the delays measured six times don't differ too much from the average value of all. In other words, that measurements are not very disperse in respect to the average of the six runs and thus the data obtained is trustworthy.

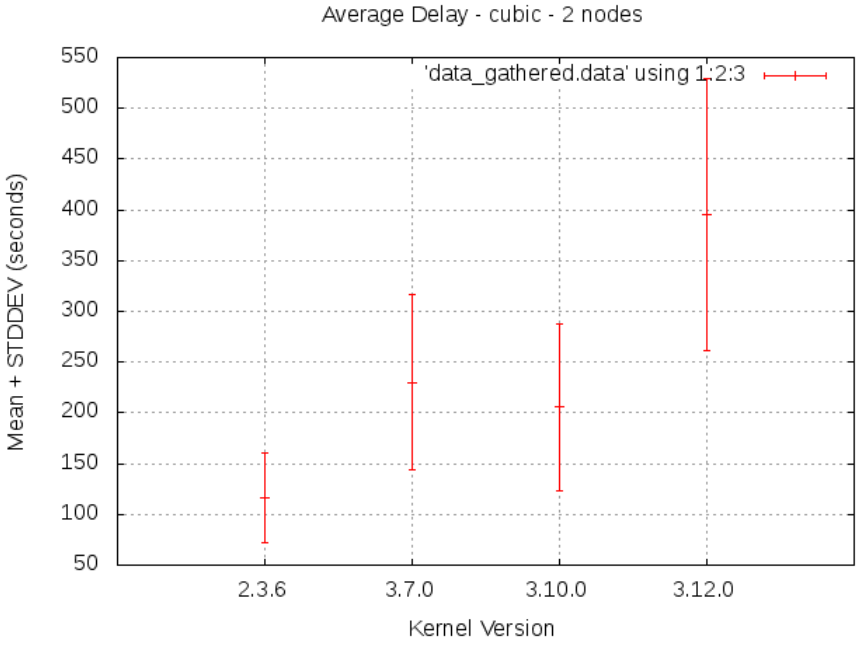

FIGURE 5.11: Average Delays Cubic 6 runs 2 nodes

Illinois, veno and yeah follow a more or less same pattern between them with some variations, but none of them stands out.
However, lp is worth being mentioned as the absolute loser of the simulations. It shows time execution delays around 200-300 seconds higher than cubic, and it's confidence intervals accross kernel versions show that measurements are pretty inaccurate. We can also confirm this in 5.12

5.2.4.2 Throughput

In this case, when congestion is forced into network's traffic, we will observe that discrepancies between throughputs of different kernel versions are even sharper (contrarily to similar results that first simulation threw).
Throughput general table will be constructed by following the same strategy as first dummy simulation, upon choosing the best pair of kernel version / average throughput for each one of the congestion control algorithms. The values obtained are the following seen in the table 5.3.

There are however some clarifications to this table that are interesting to remark, because now not only not all congestion control algorithms have similar estimates in conjunction,

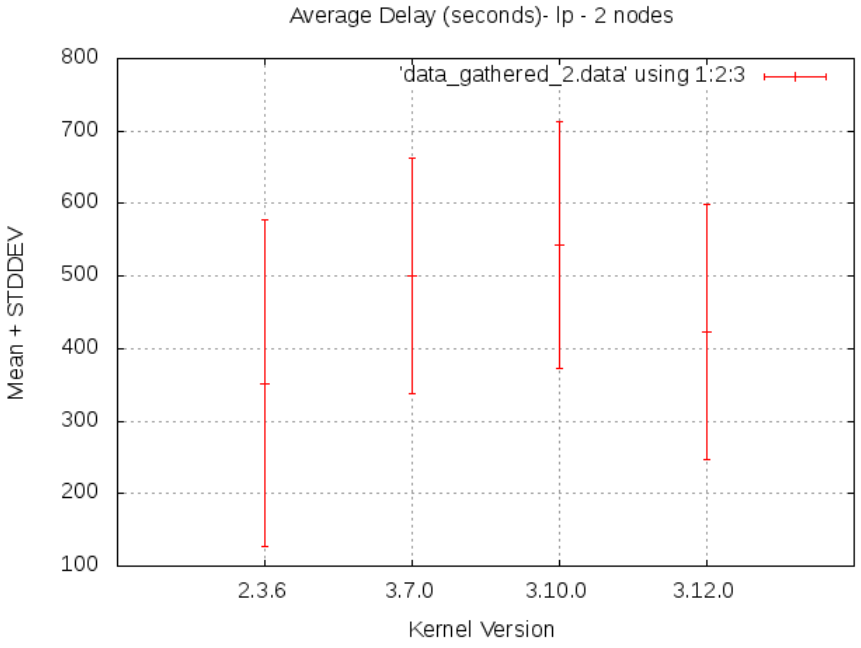

FIGURE 5.12: Average Delays Lp 6 runs 2 nodes

Algorithm	Kernel	Avg. Throughput (Kb/s)	Conf. Interval (90%)	Interval Range
cubic	2.6.36	5.602	±0.49	5.12 to 6.09
htcp	2.6.36	5.029	±0.47	4.56 to 5.5
illinois	2.6.36	4.215	±0.24	3.97 to 4.46
lp	2.6.36	3.039	±0.92	2.12 to 3.96
veno	2.6.36	3.812	±0.25	3.57 to 4.06
yeah	2.6.36	4.029	±0.35	3.68 to 4.38

TABLE 5.3: Congestion control algorithms, Kernel versions, Average Throughput 6 runs(2 nodes)

but also some of them suffer considerable discrepancies between throughputs measures in their different kernel versions.
Cubic/2.6.36 average throughput is approximately *2Kb/s* higher than the rest of kernels (not shown in table), being the smallest one in *3.12.0*, almost *3,5Kb/s* lower.
Errorbar plots are shown for those algorithms that stood out: cubic (highest), lp (lowest) and also their throughput graph to graphically see the throughput difference during it's execution in 5.13. Veno is also shown in this last one as it stands as the algorithm with the second lowest throughput, to see it's similarity with lp.

In cubic's throughput mean value graph (figure 5.14) we appreciate a stddev value of ≈ 0.5 because although the majority of throughputs for this algorithm are near *5,5Kb/s*, there has been one case with a value of $\approx 7Kb/s$, and hence the stddev, but it is a trustworthy measure.

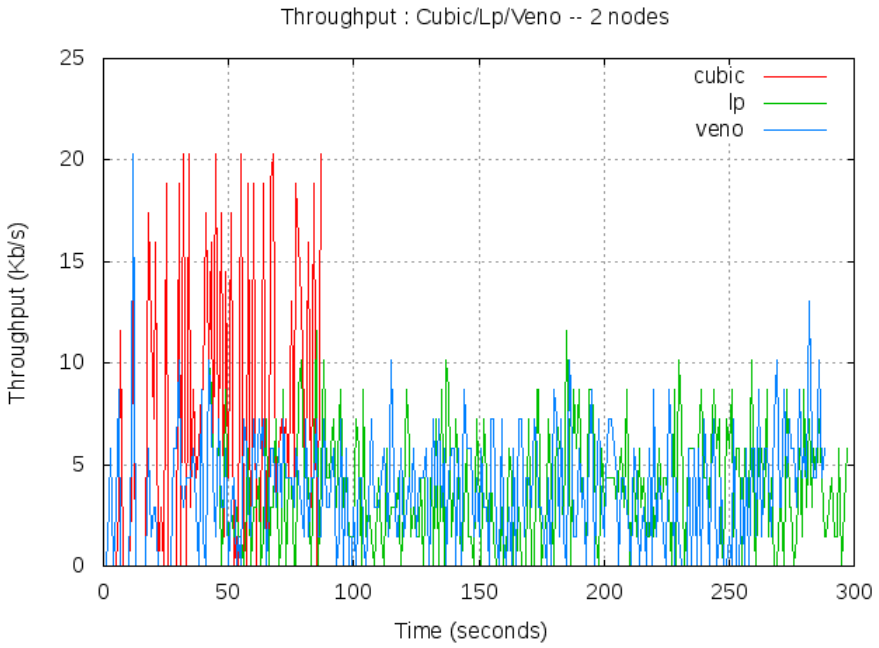

FIGURE 5.13: Throughput Cubic/Lp/Veno 2 nodes

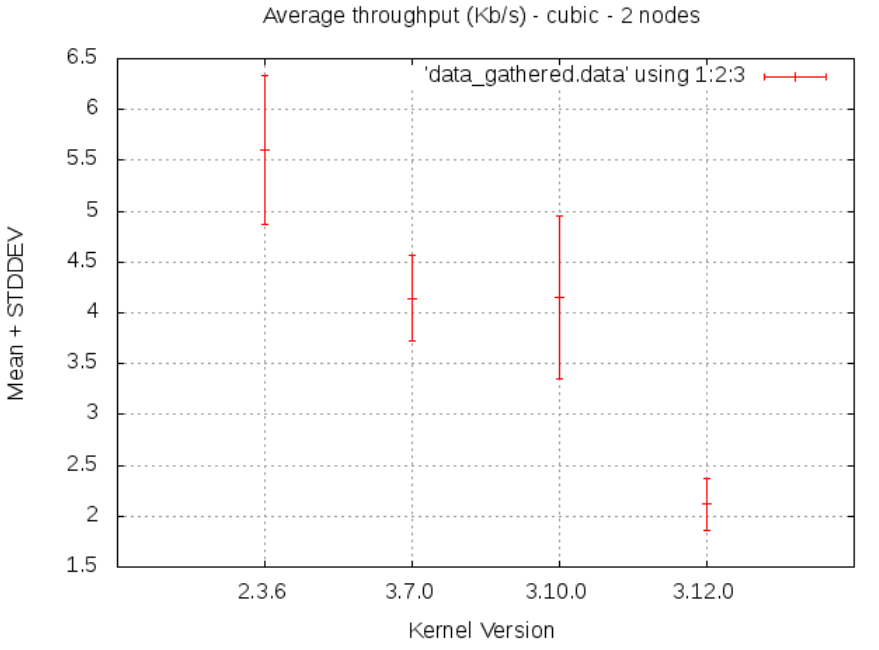

FIGURE 5.14: Throughput Mean + STDDEV Cubic 2 nodes

The slowest and worst performing algorithm, lp, shows indecisive results instead see again figure 5.15. We see that even in it's best kernel lp has the lowest throughput in comparison with the other algorithms. As we can appreciate, measures in that algorithm are really disperse. For the winner version (*2.6.36*) the stddev is about 1.5, and the rest of kernel versions are not much better. This might have something to do with the way this congestion control algorithm manages traffic.

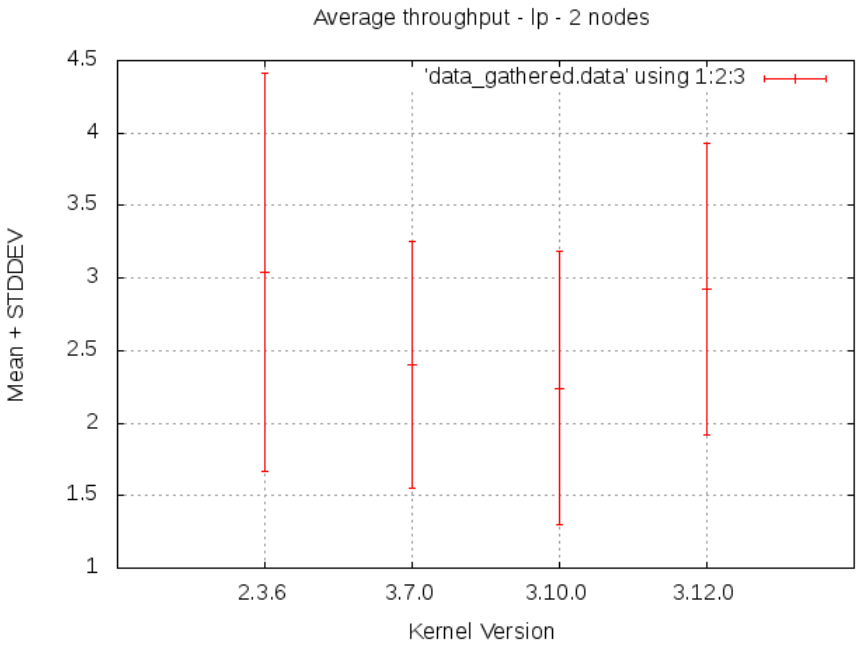

FIGURE 5.15: Throughput Mean + STDDEV Lp 2 nodes

As a general overview we see in 5.16 the average throughput values of all algorithms for the *2.6.36* version, the best performing one.

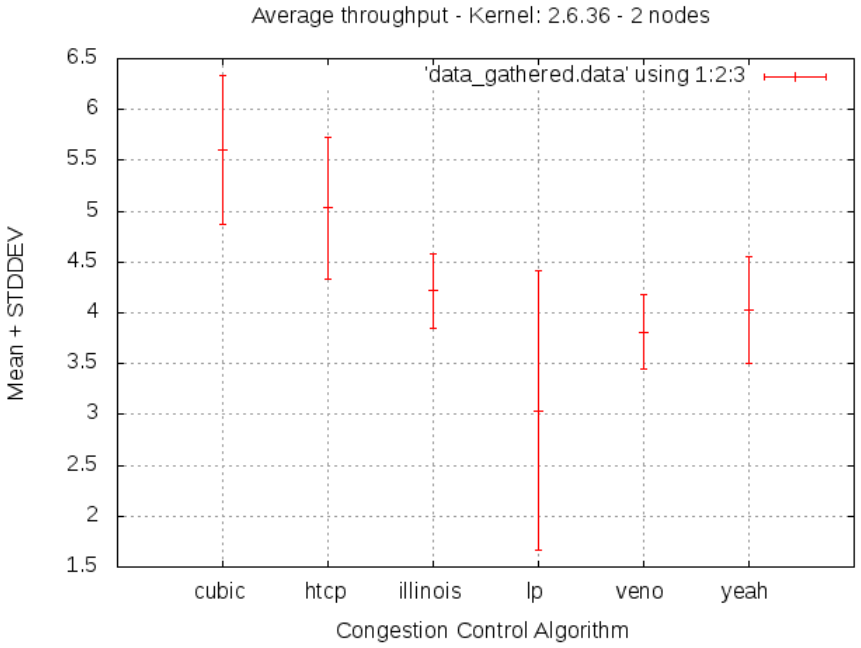

FIGURE 5.16: Throughput Mean + STDDEV all algorithms Kernel 2.6.36 - 2 nodes

5.2.4.3 Observations

LpTCP is the algorithm with the lowest throughput for a good reason. It was designed to take advantage of the leftover bandwidth when there is a situation of congestion over the network. Therefore, with two nodes it will have lower throughput than one, and we'll see that with four nodes values will be even lower.
CubicTCP has been the indiscutably winner algorithm. It was designed to deal with the problem of high Bandwidth-Delay Product (BDP). As a curiosity it is the default congestion control algorithm that linux has implemented by default. It's two main functions to manipulate the congestion window are:

$$cwnd = C * (T - K) + Max_{cwnd}$$

for the congestion window growth, and:

$$cwnd = \beta * Max_{cwnd}$$

for decreasing it, being C: a constant, T: the duration of time since last congestion event, and $K = \sqrt[3]{Max_{cwnd} * \beta / C}$. It aims to provide equal fairness to all the competing tcp connections that are also in the network, aswell with other improvements such as:

- Reduction of RTT bias.
- Rapid recovery of cwnd following its decrease due to a loss event, maximizing throughput.
- Rapid expansion of cwnd when a raised network ceiling is detected.

Cubic allows very fast window expansion, however, it also makes attempts to slow the growth of cwnd sharply as cwnd approaches the current network ceiling. We can see how it performs against other algorithms in other studies such as it's introducing paper: [9]. As seen, there are many variables that come into play when it comes to decide how cubic behaves, and testing each of them would be a topic for a series of isolated experiments themselves. It suffices to say that in general lines it performs well in a wide range of topologies and situations, as it is the default choice for nowadays machines.
Respect to kernel version performances, it is safe to think that in this case where the simplest flow-competing scenario with two nodes fighting for available bandwidth takes place, *2.6.36* has proven to be in general lines the best performing one. And, in contraposition, *3.12.0* throws the worst results among all versions. Linux changelog for *2.6.36* shows that not only no specific new networking improvements were added, but checking it's tcp related sysctl parameters it lacks some of the TCP improvements the

other versions possess: it lack's *Forward acks (f-ack)*, *Early retransmit*, *Tail-Loss Probes (tlp)*, and the newest *Fast Open*. *3.12.0* version reunites all those tcp extensions and for some reason, within the same congestion control algorithm, results are much worse.

5.2.5 Results: 4 Nodes

5.2.5.1 Delays

Following the same line as the 2 node case, a table with the average delay times for each pair algorithm/kernel version and the confidence interval with a confidence factor of 90% is provided, we can see it in 5.4

Algorithm	Kernel Version	Avg. Delay(seconds)	Conf. Interval (90%)	Interval Range
cubic	2.6.36	283	±103.31	179.84 to 386.46
	3.7.0	308	±125.16	183.74 to 434.05
	3.10.0	393	±22.5	371.16 to 416.16
	3.12.0	582	±145.25	437.47 to 727.98
htcp	2.6.36	314	±110.11	204.49 to 424.7
	3.7.0	452	±58	394.66 to 510.65
	3.10.0	461	±35.5	425.69 to 496.68
	3.12.0	464	±142.03	322.05 to 606.1
illinois	2.6.36	424	±71.71	353.28 to 496.71
	3.7.0	423	±99.49	324.19 to 523.16
	3.10.0	429	±19.21	410.61 to 449.02
	3.12.0	472	±127.7	344.56 to 599.97
lp	2.6.36	1176	±283.56	892.76 to 1459.88
	3.7.0	911	±338.59	572.5 to 1249.67
	3.10.0	1320	±264.02	1056.9 to 1584.95
	3.12.0	1256	±405.66	851.09 to 1662.41
veno	2.6.36	519	±46.23	472.82 to 565.29
	3.7.0	486	±36.97	449.59 to 523.53
	3.10.0	490	±52.75	437.43 to 542.94
	3.12.0	547	±48.91	498.38 to 596.19
yeah	2.6.36	467	±70.46	397.01 to 537.94
	3.7.0	417	±130.53	286.5 to 547.56
	3.10.0	408	±56.3	352.64 to 465.25
	3.12.0	503	±37.4	566.37 to 641.17

TABLE 5.4: Cong. control algorithms, Avg delays and statistics, 4 nodes

We see that the situation for 4 nodes looks close to the 2 node case but with a significant increase of time for all connections due to the addition of network congestion and complexity: lp as the absolute slowest algorithm really far away from the others, cubic as the winner again and veno standing up as the second slowest one, but this time with values much closer between all algorithms.

In this scenario the rest of the algorithms' times tend to overlap more, there is definitely no clear winner among them. Some kernel versions are faster than others but values are close between them. Only version *2.6.36* again like with the 2-node case is slightly faster in all cases. Once again is necessary to verify, *are delay measurements accurate?* we will check it for the winner algorithm and the indiscutably loser. Cubic in 5.17 and lp in 5.18.

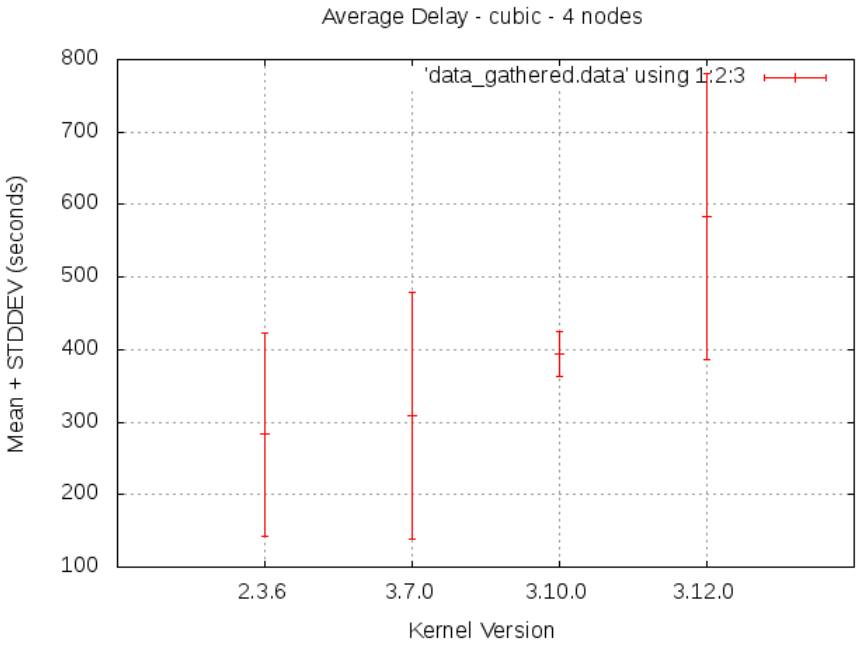

FIGURE 5.17: Average Delays Cubic 6 runs 4 nodes

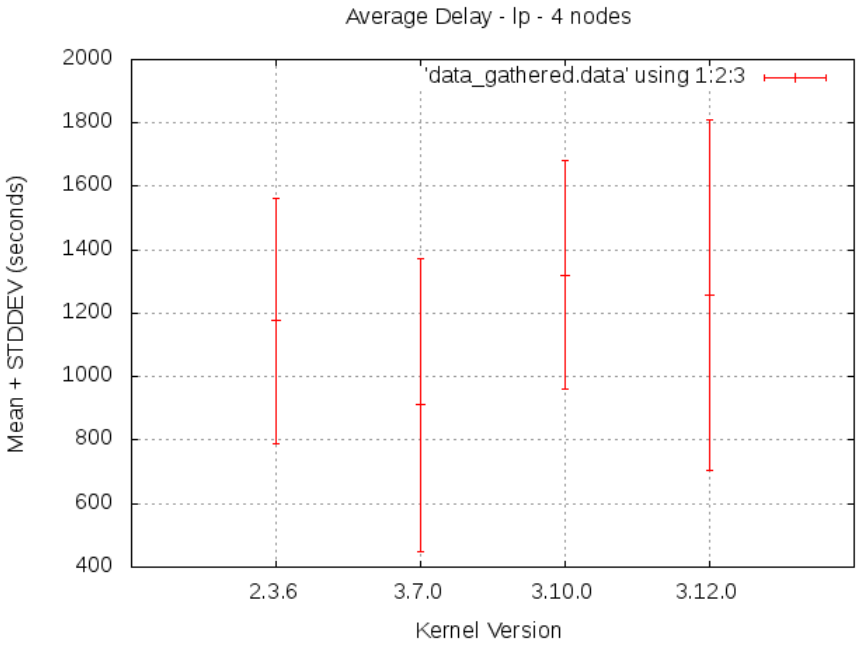

FIGURE 5.18: Average Delays Lp 6 runs 4 nodes

Cubic's smallest standard deviation is in kernel version *3.10.0*. Despite that rest of the kernel versions have alike values for this measure and therefore cubic's measurements could be considered valid for this study, it would take a bigger sample to confirm or discard any further assumptions. Cubic's measurements also share a confidence interval that varies around 100 seconds. Rest of the algorithms exceptuating lp have lower confidence intervals so we deduce that it is some simulator issues and it would take more runs to have a more accurate average. Lp data has a confidence interval that reaches up to ±400 seconds in the worst case and ±260 seconds in the best. Lp's behaviour is somehow difficult to narrow.

5.2.5.2 Throughput

As in the two node simulation, if we showed throughput graph for all joint algorithm's nothing could be distinguished between so many overlapping functions. This table (congestion algorithm / kernel version) will properly show winning pairs. (table in 5.5).

Algorithm	Kernel	Avg. Throughput (Kb/s)	Conf. Interval (90%)	Interval Range
cubic	3.10.0	2.78	±0.15	2.62 to 2.92
htcp	2.6.36	2.78	±0.42	2.36 to 3.2
illinois	2.6.36	2.684	±0.3	2.38 to 2.99
lp	3.12.0	1.084	±0.36	0.72 to 1.45
veno	3.10.0	2.252	±0.27	1.99 to 2.52
yeah	3.10.0	2.732	±0.36	2.37 to 3.1

TABLE 5.5: Congestion control algorithms, Kernel versions, Average Throughput 6 runs (4 nodes)

The table shows that throughputs are uniformly lower, this is normal because bandwidth is now being shared among more flows.
In this scenario there is no clear winner, all of them but lp share almost identical throughput values. Cubic, htcp and yeah are practically equal, being illinois and veno a little below them but could be easily due to the randomness incorporated to the simulations. Lp as the previous simulation shows an incredibly low throughput (more than a half lower) in comparison to the others.
As a more visual example, in 5.19 we see cubic, htcp and illinois throughput's practically overlap. Cubic shows more execution time than the others because the highest throughput value for this algorithm is found in version *3.10.0*, while the lowest delay value is found in *2.6.36*. 5.20 compares cubic throughput with lp.

From the throughput's table (5.5) it is safe to deduce that in general lines there is no best performing kernel. Although values printed in the table correspond to the higher

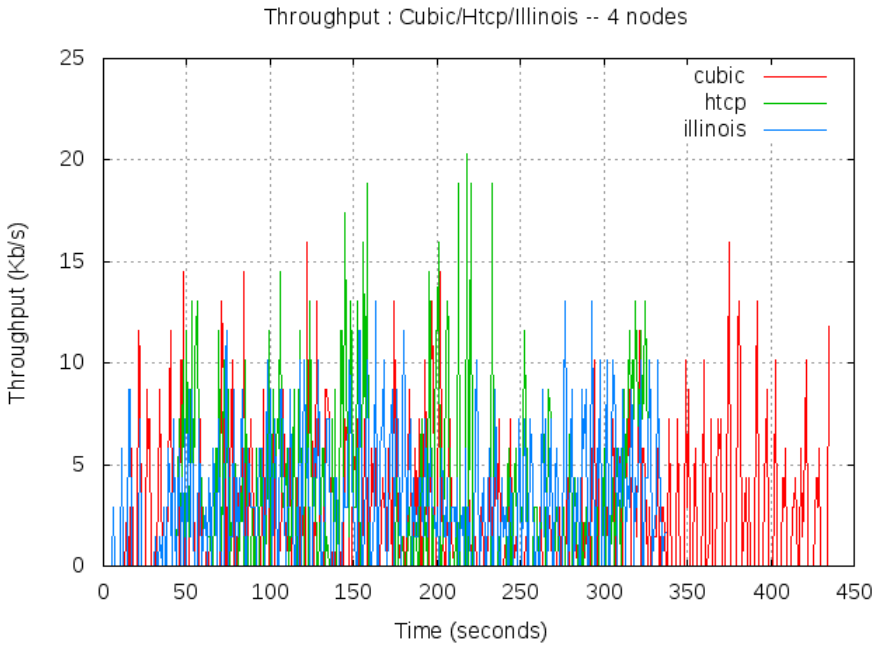

FIGURE 5.19: Throughput Cubic/Htcp/Illinois 4 nodes

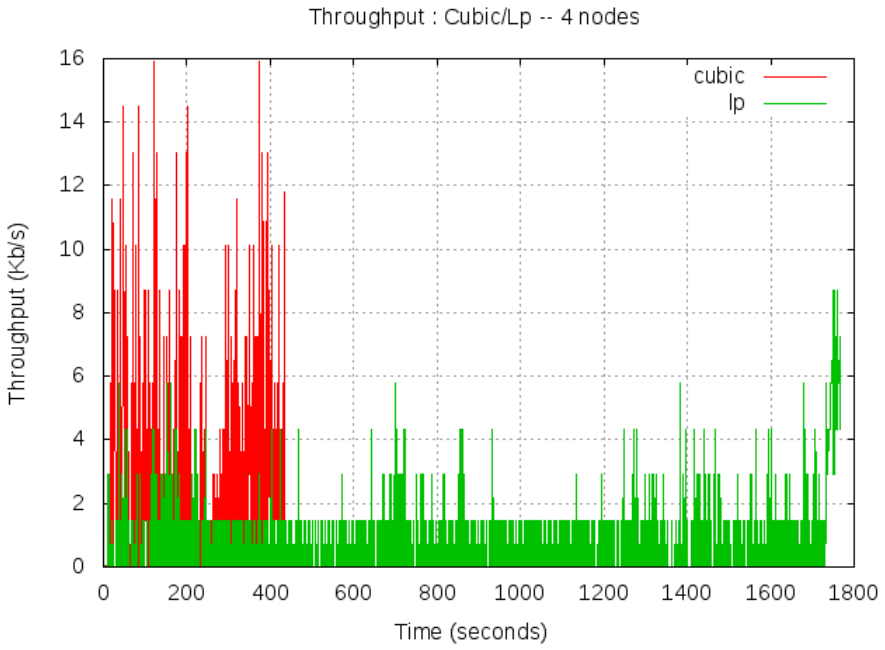

FIGURE 5.20: Throughput Cubic/Lp 4 nodes

average throughput of an algorithm and the kernel version with which it was achieved, numbers don't differ too much from the other kernel versions. Measures taken in this scenario are trustworthy. Standard deviation for cubic is only ± 0.2 for a throughput value of *2.78*, while the highest deviation can be found in htcp with a value of ± 0.6, which is still acceptable. In 5.21 we graphically see the pairs algorithm/kernel combined in an error bar plot.

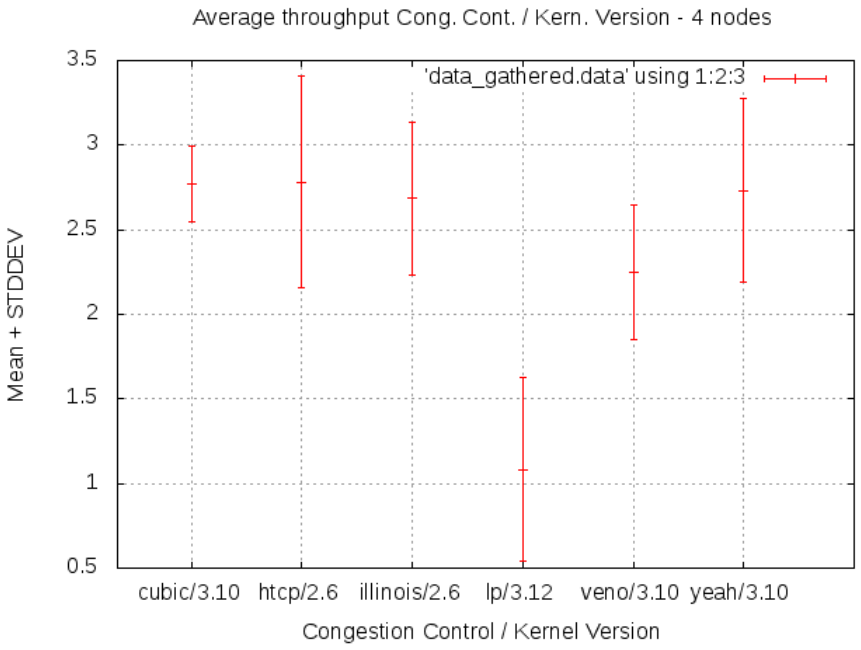

FIGURE 5.21: Throughput Mean + STDDEV All algorithms 4 nodes

5.2.5.3 Observations

Execution delays are proportionally higher, and throughput values are proportionally lower, as available bandwidth must be distributed between competing flows. Now, it is specially interesting to see the inflight data graphics of those algorithms that attracted the most attention during the course of the simulations, to see how they manage the way to send packets during the connection. In this case it can be interesting to see the differences on how *cubic/3.10.0* and *lp/3.12.0* manage the congestion window.
We can see the graphic for cubic in 5.22 and lp in 5.23.

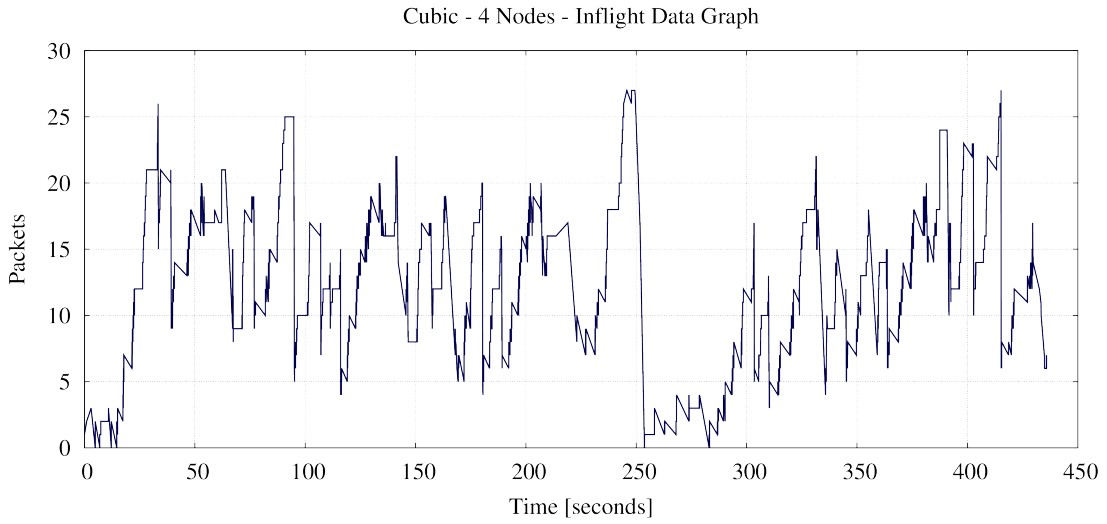

FIGURE 5.22: Inflight Data Cubic 4 nodes

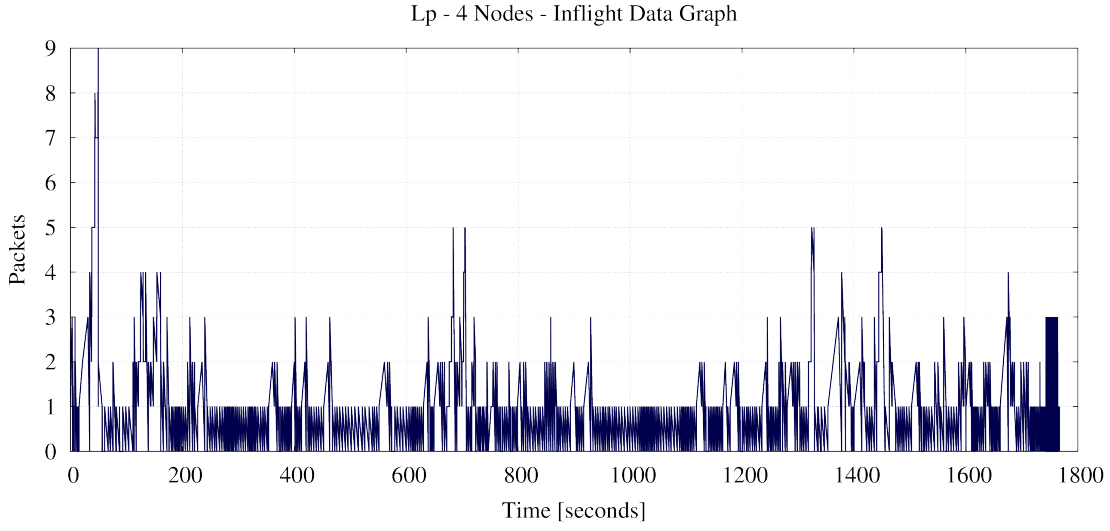

FIGURE 5.23: Inflight Data Lp 4 nodes

Cubic (figure 5.22), it's a little hard to interpret. Apparently it follows quite a conservative behaviour, by increasing it's congestion window rapidly during the start of the connection, and then cutting low and re-increasing the cwnd in order to cope with a congested link (in this case, put there on purpose).
It is perfectly evident in figure 5.23 how lp maintains a very low throughput by keeping the rate of sent data segments at the minimum. Regarding kernel versions, in terms of throughput there is definitely no clear winner. Results shown in 5.5 are merely orientative, as for every algorithm all kernel versions casted really similar results.
However, when looking at the delay execution time in 5.4 there is one kernel version that falls behind all it's competitors: *3.12.0*. Newest kernel versions is either in 5.1 and 5.2 the worst performing one by far.

5.2.6 Results: 6 Nodes

5.2.6.1 Delays

Here is the table with the average delay times in this six node case, we see it in 5.6.

Results definitely follow the line of previous scenarios. Cubic is the clear winner being the best kernel version *2.6.36* either with this algorithm and with the other ones and lp ranks last. Only in a couple cases (for sure due to the randomness of the simulations) version *2.6.36* is not the distinctive winner by providing the lowest delay execution times (see illinois or yeah), but in general lines there is the best performing one.
In 5.24 we appreciate how *2.6.36* gives the smallest delay and the standard deviation is not very disparate among versions (≈ 100), which is quite acceptable.

It is however in lp, as in previous simulations where we find either in 5.6 or graphically in 5.25 the most distorted measurements, with the highest standard deviation.

5.2.6.2 Throughput

This simulation's average throughputs can be seen in 5.7.

Although in illinois and yeah the highest results were achieved by kernel *3.7.0*, throughput measurements for the first three versions practically don't differ, hence, it is safe to declare version *2.6.36* as the best performing one generally for all congestion control algorithms.
Once again, *cubic/2.6.36* shows the best results, and can be graphically compared with the other versions in the errorbar plot in 5.26. Although the winning version shows the highest standard deviation, it is acceptable when compared with the rest of algorithm's.

Algorithm	Kernel Version	Avg. Delay(seconds)	Conf. Interval (90%)	Interval Range
cubic	2.6.36	402	±110.04	292.11 to 512.19
	3.7.0	424	±103.32	321.12 to 527.76
	3.10.0	544	±85.32	458.51 to 629.14
	3.12.0	748	±107.03	640.69 to 854.75
htcp	2.6.36	476	±95.06	381.04 to 571.17
	3.7.0	593	±113.87	479.61 to 707.34
	3.10.0	521	±66.19	454.88 to 587.26
	3.12.0	761	±104.14	656.8 to 865.09
illinois	2.6.36	682	±107.22	574.73 to 789.16
	3.7.0	644	±106.64	537.6 to 750.88
	3.10.0	622	±103.7	518.45 to 725.86
	3.12.0	827	±61.02	766.24 to 888.29
lp	2.6.36	1152	±259.17	893.1 to 1411.45
	3.7.0	1842	±422.68	1419.41 to 2264.78
	3.10.0	1751	±416.01	1335.78 to 2167.8
	3.12.0	1986	±487.4	1498.32 to 2473.12
veno	2.6.36	701	±62.59	638.25 to 763.43
	3.7.0	778	±83.63	694.65 to 861.92
	3.10.0	719	±62.7	656.32 to 781.72
	3.12.0	903	±143.39	759.96 to 1046.73
yeah	2.6.36	601	±47.12	554.1 to 648.34
	3.7.0	544	±56.77	487.5 to 601.05
	3.10.0	570	±52.44	517.15 to 622.03
	3.12.0	820	±80.83	739.4 to 901.06

TABLE 5.6: Cong. control algorithms, Avg delays and statistics, 6 nodes

Algorithm	Kernel	Avg. Throughput (Kb/s)	Conf. Interval (90%)	Interval Range
cubic	2.6.36	2.586	±0.43	2.15 to 3.02
htcp	2.6.36	2.059	±0.37	1.69 to 2.43
illinois	3.7.0	1.822	±0.32	1.5 to 2.14
lp	2.6.36	1.045	±0.22	0.83 to 1.26
veno	2.6.36	1.614	±0.13	1.48 to 1.75
yeah	3.7.0	2.042	±0.18	1.86 to 2.23

TABLE 5.7: Congestion control algorithms, Kernel versions, Average Throughput 6 runs (6 nodes)

Last figure (5.27) shows the errorbar plot for all pairs of congestion control algorithm and kernel version that stood out as winners in this scenario.

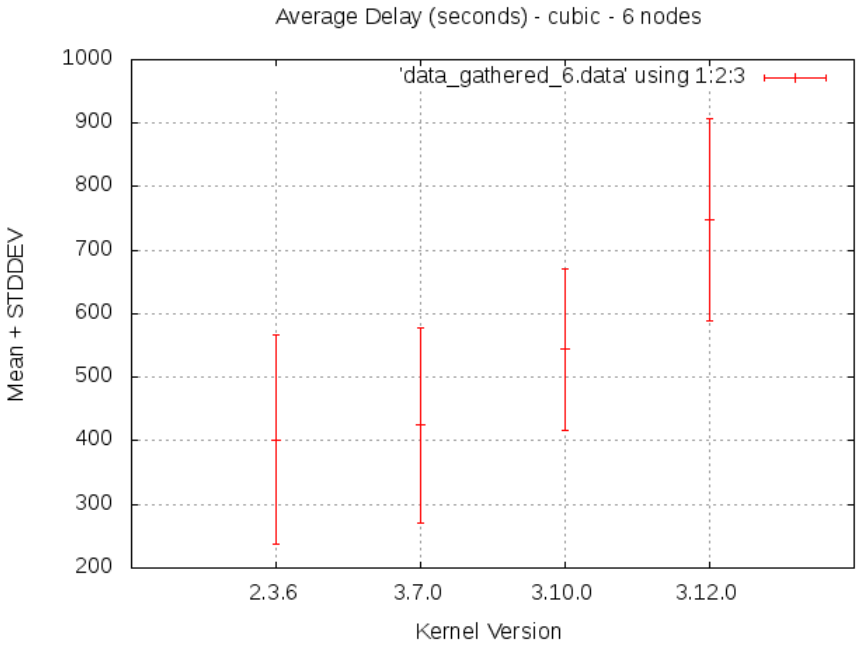

FIGURE 5.24: Average Delays Cubic 6 runs 6 nodes

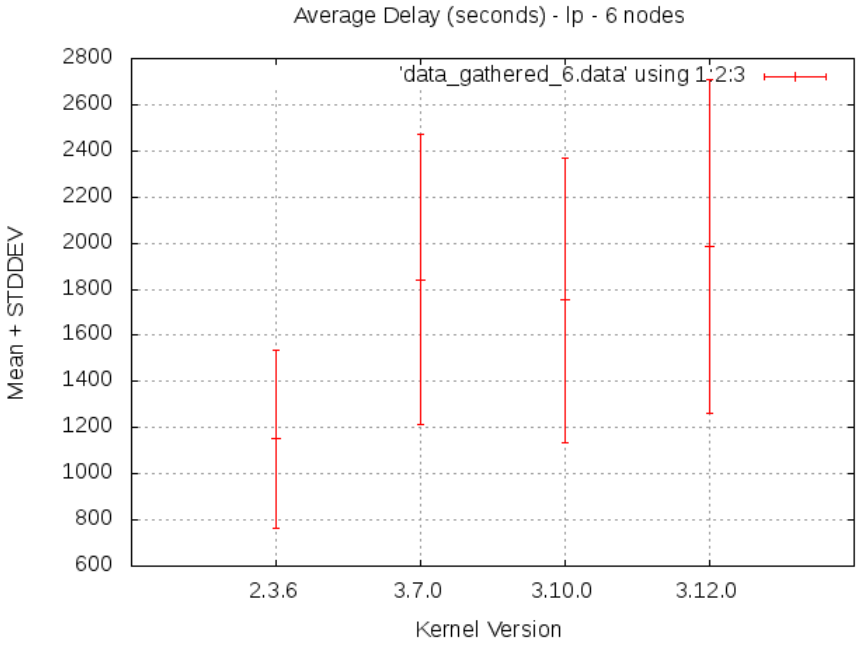

FIGURE 5.25: Average Delays Lp 6 runs 6 nodes

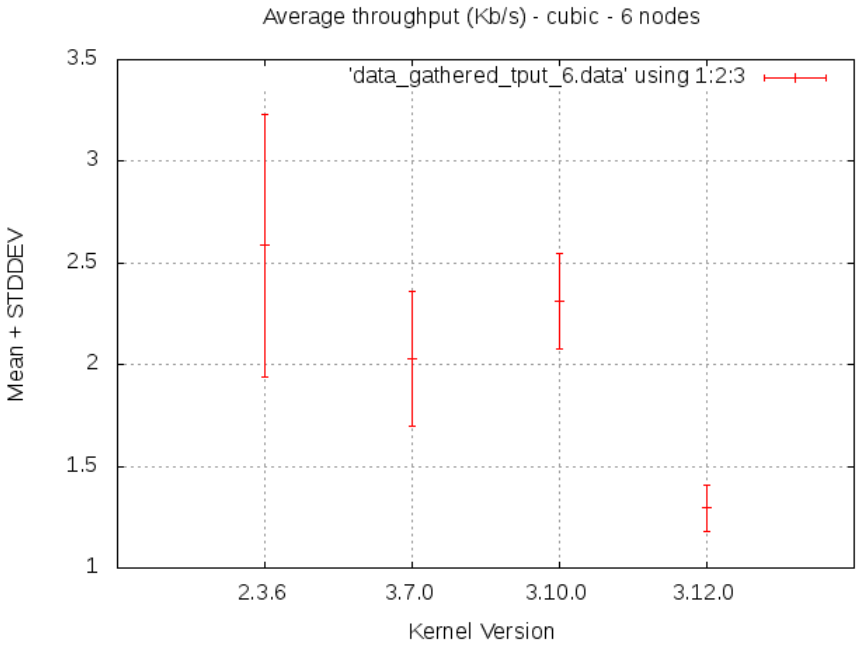

FIGURE 5.26: Throughput Cubic 6 nodes

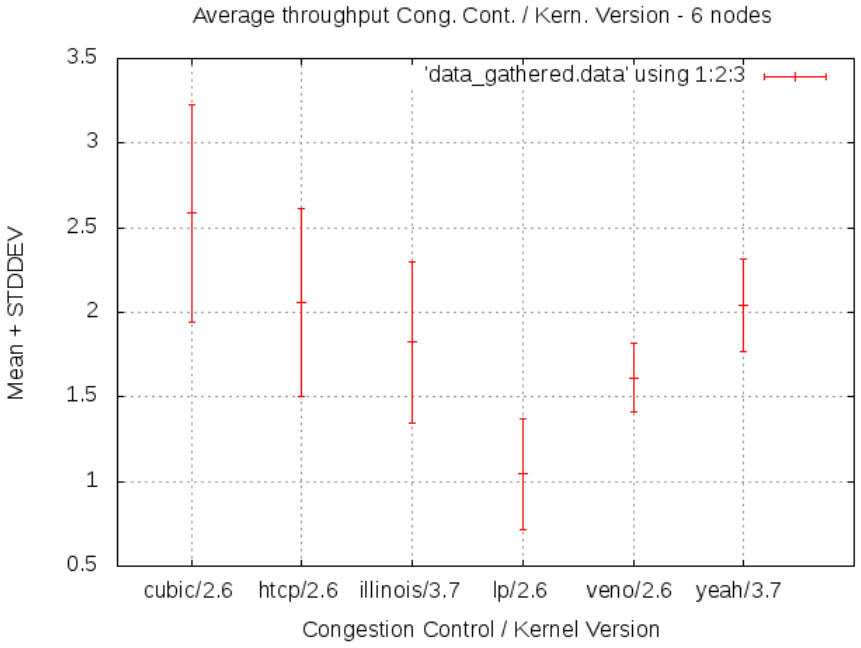

FIGURE 5.27: Throughput Mean + STDDEV All algorithms 6 nodes

5.2.6.3 Observations

Results are similar to the two and four node scenarios, being execution delays proportionally higher and throughput values proportionally lower, due to the newly incorporated flows that compete for network resources. As a dummy rule of thumb, delay values appear to be approximately 100 seconds higher every time two new competing flows are added to the client side.
Whilst delay values follow that approximate rule, throughput values are in general slightly lower when we add new flows but don't obey to any particular rule. To do so it would most likely be needed to study each pair algorithm & kernel version sepparately and adding more executions to have a bigger sample size.
Regarding kernel versions, the only for-sure assertion is that *3.12.0* still performs the worst.

5.2.7 Results: 8 Nodes

5.2.7.1 Delays

The table 5.8 shows the average delay data for this scenario.

Delay execution times are around 100 seconds slower than six node scenario, which behaved also in the same manner in respect to the four node one.
It is however surprising how the best performing algorithm is *htcp/2.6.36* instead of cubic. While taking a look at both's algorithms measurements log files we see that simulation results oscilate between 200 and 600 seconds, really disparate measures, so it is normal that htcp would easily have better times. Standard deviation for cubic is ≈ 151 and for htcp is ≈ 214, meaning that values for htcp are up to 60 seconds above and below the average, making the fact of htcp having a lower delay execution time than cubic possible, but not to be trust in the long term. More simulations with a much bigger sample should be done in order to break the tie.
For cubic, htcp and illinois kernel version *2.6.36* gathers the best results. For lp it is version *3.7.0* and veno and yeah share the best times with *3.10.0*. Even kernel *3.12.0* does not give generally disastrous results as in previous simulations, hence there is no clear winner version in this case.
Only one thing remain unchanged: lp is the slowest algorithm.

5.2.7.2 Throughput

Throughput values for this scenario can be found in 5.9.

Algorithm	Kernel Version	Avg. Delay(seconds)	Conf. Interval (90%)	Interval Range
cubic	2.6.36	534	±101.85	432.17 to 635.87
	3.7.0	565	±43.72	521.22 to 608.67
	3.10.0	652	±107.68	544.54 to 759.9
	3.12.0	1083	±41.56	1041.42 to 1124.54
htcp	2.6.36	401	±144.14	256.96 to 545.24
	3.7.0	572	±96.25	475.49 to 667.98
	3.10.0	646	±68.9	577.39 to 715.19
	3.12.0	1085	±106.75	977.97 to 1191.46
illinois	2.6.36	758	±119.89	637.7 to 877.48
	3.7.0	881	±107.49	773.32 to 988.3
	3.10.0	798	±151.62	645.89 to 949.14
	3.12.0	880	±125.12	754.6 to 1004.83
lp	2.6.36	2059	±169.51	1889.04 to 2228.05
	3.7.0	1923	±448.99	1473.79 to 2371.79
	3.10.0	2208	±200.39	2007.16 to 2407.94
	3.12.0	2403	±309.67	2093.74 to 2713.09
veno	2.6.36	1036	±74.11	962.16 to 1110.38
	3.7.0	968	±138.57	829.91 to 1107.04
	3.10.0	870	±63.63	806.42 to 933.69
	3.12.0	995	±100.21	895.1 to 1095.52
yeah	2.6.36	960	±217.69	742.3 to 1177.68
	3.7.0	750	±83.82	666.56 to 834.2
	3.10.0	742	±64.31	677.29 to 805.91
	3.12.0	930	±31.03	899.08 to 961.14

TABLE 5.8: Cong. control algorithms, Avg delays and statistics, 8 nodes

Algorithm	Kernel	Avg. Throughput (Kb/s)	Conf. Interval (90%)	Interval Range
cubic	2.6.36	2.249	±0.57	1.68 to 2.81
htcp	2.6.36	2.227	±0.36	1.87 to 2.59
illinois	2.6.36	1.556	±0.24	1.31 to 1.8
lp	3.7.0	0.656	±0.28	0.37 to 0.94
veno	3.10.0	1.290	±0.09	1.2 to 1.38
yeah	3.7.0	1.543	±0.16	1.38 to 1.7

TABLE 5.9: Congestion control algorithms, Kernel versions, Average Throughput 6 runs (8 nodes)

Throughputs are more alike the previous simulations. *Cubic/2.6.36* has the highest throughput value of all algorithms (although the difference with htcp is only but negligible). Lp in the kernel version *3.7.0* stands as the worst performing one, and rest of them behave equally poorly. Checking the errorbar plot for cubic in 5.28 we see that the *2.6.36* version offers an incredibly high standard deviation respect to the other versions. This is because in the throughput measurement log there are several measures that exceed the mean with a relatively high value, making it's average throughput spike

up.

Now we check htcp error bar plot in 5.29 and again, the *2.6.36* version has the highest mean value but the highest standard deviation too. The standard deviation value for *cubic/2.6.36* is ≈ 0.8 and *htcp/2.6.36*'s one is ≈ 0.5, hence, htcp measurements are not enough to make cubic discarded in htcp's favor. It would be needed a much bigger sample of simulations in order to be sure about it.

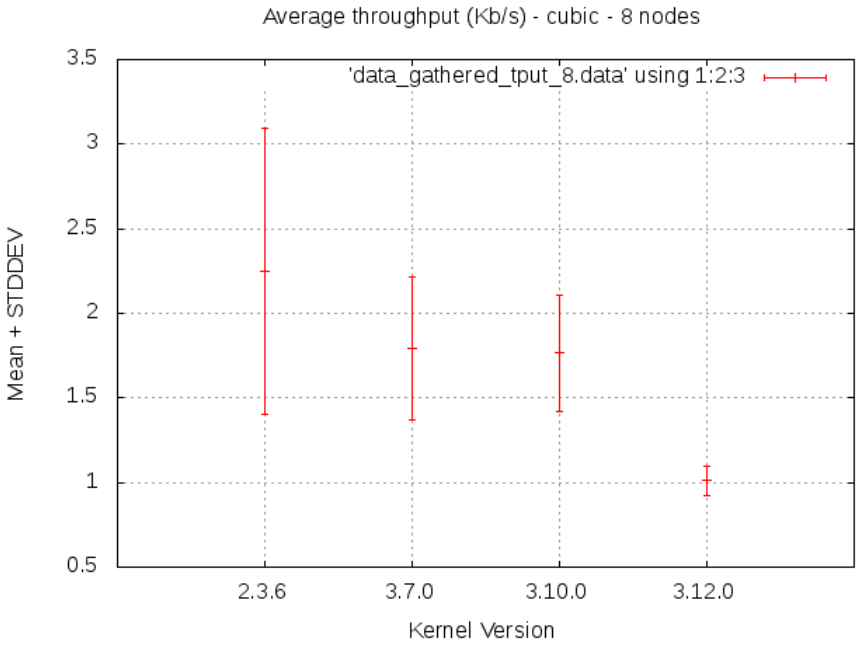

FIGURE 5.28: Throughput Cubic 8 nodes

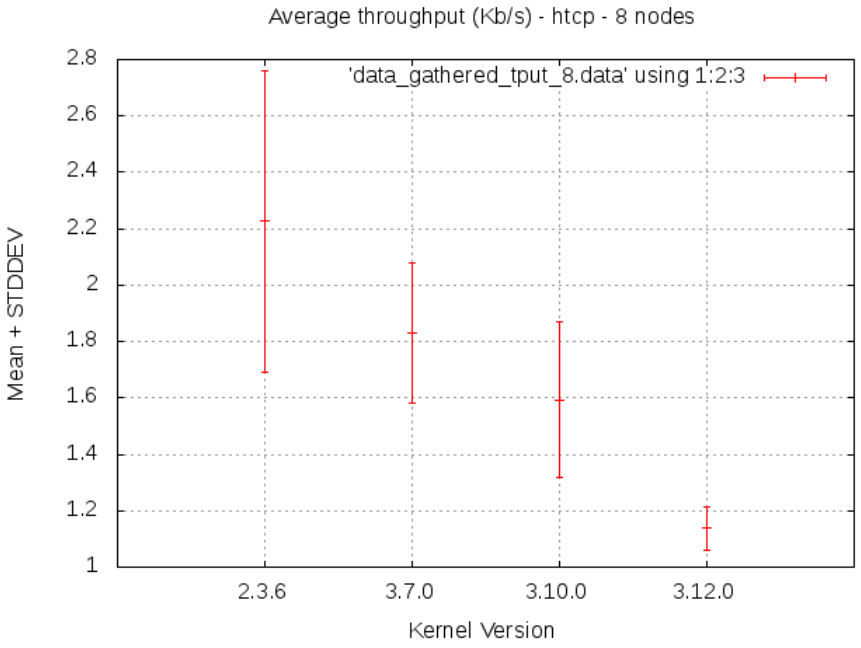

FIGURE 5.29: Throughput Htcp 8 nodes

5.2.7.3 Observations

It definitely seems that with 8 nodes in client side, cubic and htcp performances start to overlap and behave in a similar way. Both of them offer near delay and throughput values, and when checking their measurement logs they tend to throw disparate values for every simulation either with delays and throughputs, therefore the shared high standard deviation. Lp is as always the slowest and with lowest throughput, and the rest of the algorithms cast more or less equal results.
Table 5.10 shows a joint table of all algorithms and their corresponding smallest average delays in each one of the scenarios measured. Kernel versions are not shown because sometimes they change accross scenarios and so we will assume the winning one for each, as now we are just paying attention to check if any patterns are found.

Algorithm	delay 2 nodes(s)	delay 4 nodes(s)	delay 6 nodes(s)	delay 8 nodes(s)
cubic	115	283	402	534
htcp	166	314	476	401
illinois	232	423	622	758
lp	352	911	1152	1923
veno	256	486	701	870
yeah	224	408	544	742

TABLE 5.10: Congestion control algorithms, Average delays for all scenarios

We see in 5.10 that when it comes to delay execution time, values are ≈ 150 higher every time two new competing flows are added to the client side, exceptuating lp which grows exponentially every new simulation. When looking at htcp's case for 8 nodes the results are surprising, it definitely breaks the overall pattern and gives a slightly better time than it's previous case, therefore it must have been a problem with the simulator. The same table is shown for throughputs in 5.11.

Algorithm	tput 2 nodes(Kb/s)	tput 4 nodes(Kb/s)	tput 6 nodes(Kb/s)	tput 8 nodes(Kb/s)
cubic	5.602	2.780	2.586	2.249
htcp	5.029	2.780	2.059	2.227
illinois	4.215	2.689	1.822	1.556
lp	3.039	1.089	1.045	0.656
veno	3.812	2.252	1.614	1.290
yeah	4.029	2.732	2.042	1.543

TABLE 5.11: Congestion control algorithms, Average throughputs for all scenarios

Throughput does not follow a specific pattern when competing flows are increased. In general when passing from 2 to 4 nodes throughput is reduced in ≈ 2.5 for most algorithms, but in subsequent cases the addition of new nodes to the client side does not

necessarily mean a linear decrease. Indeed what goes for granted is actual throughput reduction, but not actually following any pattern.
When looking from the basic 2 node case to the final with 8 nodes, doing an approximate substraction:

- cubic: 5.6-2.2 = 3.4
- htcp : 5-2.2 = 2.8
- illinois: 4.2-1.5 = 2.7
- lp: 3-0.6 = 2.6
- veno: 3.8-1.2 = 2.6
- yeah: 4-1.5 = 2.5

We see that as a general rule throughput is reduced in *3 Kb/s* from the initial setup to the last simulation performed.

Chapter 6

Conclusions

TCP has been originally designed to provide a reliable, in order and error free information transmission protocol. As technologies and networks have been growing and evolving, TCP has been forced to go and evolve with them, so many different extensions, patches and changes have been proposed and implemented to improve it. On each new release, the Linux Kernel has adopted some of those features in it's TCP stack.
This thesis conducts a set of simulations that aim to provide a general approach and overall insight on how four different implementations of the TCP stack, corresponding to distinct Linux Kernel releases, behave when we vary an integral and key piece of it's inner workings: the congestion control algorithm. To do so, two different main topologies are established:

1. A simple single flow topology consisting of a client/server approach with two middle routers. This case serves as a working basis in order to scale it afterwards.

2. A multi-flow topology where several clients attempt to sent data at the same time to a single server with two middle routers with limited queues, therefore providing a good example of a situation of network congestion.

Simulations are performed using the well known academic ns3 network simulator, in conjunction with the DCE framework, that allows to execute native code inside the context of the simulator, and hence permitting us to execute the actual Linux Kernel's TCP stack code.
To do so, two main parameters were chosen as they stand as good indicators of a tcp connection's performance: goodput (actual useful data that goes through a link, don't confuse with throughput), and execution delay time.
This thesis has it's limitations. The heterogeneity on network topologies, protocols and means of communication reduces this study to the analysis of a small group of real world

examples. In addition, some of the tools used are research tools and therefore subject to failure, however results shown in this piece of work are proven to be consistent over multiple tests.

This document provides a testbed of the most well known congestion control algorithms executed accross multiple versions of the linux kernel, and they may serve as a good starting point for researchers to have a general idea on how they perform when they are all put through the same specific scenario. The most surprising idea around it's findings is that performance of a specific congestion control algorithm is not strictly bound to the latest and supposedly more tuned and fixed kernel version, but instead in general lines those algorithms show better results in the oldest kernel version tested.

Further work based on the concepts handled on this thesis is easy to scale: new topologies can be added (star, ring, data center), new media access can be emulated (LTE), new tools can be used to test other ways rather than bulk data transfer (i.e. netperf, with request/response), even many other congestion control algorithms not contemplated during this thesis can be added to the simulations (AfricaTCP, DataCenter TCP (DCTCP)).

Bibliography

[1] Jawhar Abed et al. "Comparison of High Speed Congestion Control Protocols". In: *International Journal of Network Security and Its Applications (IJNSA)* 4.5 (2012). URL: airccse.org/journal/nsa/0912nsa02.pdf.

[RFC5827] M. Allman. *Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)*. RFC5827. Apr. 2010. URL: https://tools.ietf.org/html/rfc5827.

[RFC3042] M. Allman. *Enhancing TCP's Loss Recovery Using Limited Transmit*. RFC3042. Jan. 2001. URL: https://tools.ietf.org/html/rfc3042.

[RFC3390] M. Allman. *Increasing TCP's Initial Window*. RFC3390. Sept. 2002. URL: https://tools.ietf.org/html/rfc3390.

[RFC5681] M. Allman. *TCP Congestion Control*. RFC5681. Sept. 2009. URL: https://tools.ietf.org/html/rfc5681.

[2] Bake. *Bake, An integration tool for ns3*. URL: https://planete.inria.fr/software/bake/index.html.

[RFC6675] E. Blanton. *A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP*. RFC6675. Aug. 2012. URL: https://tools.ietf.org/html/rfc6675.

[RFC7323] D. Borman. *TCP Extensions for High Performance*. RFC7323. Sept. 2014. URL: https://tools.ietf.org/html/rfc7323.

[3] Carlo Caini and Rosario Firrincieli. "Tcp hybla: a tcp enhancement for heterogeneous networks". In: *INTERNATIONAL JOURNAL OF SATELLITE COMMUNICATIONS AND NETWORKING* 22 (2004).

[draftfastopen] Y. Cheng. *TCP Fast Open*. draftfastopen. June 2012. URL: https://tools.ietf.org/html/draft-ietf-tcpm-fastopen-01.

[RFC7413] Y. Cheng. *TCP Fast Open*. RFC7413. Dec. 2014. URL: https://tools.ietf.org/html/rfc7413.

[4] Jonathan Corbet. *TCP Small Queues*. 2012. URL: https://lwn.net/Articles/507065/.

[RFC7414] M. Duke. *A Roadmap for Transmission Control Protocol (TCP)*. RFC7414. Feb. 2015. URL: https://www.rfc-editor.org/rfc/rfc7414.txt.

[drafttlp] N. Dukkipati. *Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses*. drafttlp. Feb. 2013. URL: https://tools.ietf.org/html/draft-dukkipati-tcpm-tcp-loss-probe-01.

[5] Eric Dumazet. *tcp: TCP Small Queues*. 2012. URL: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=46d3ceabd8d98ed0ad10f20c

[RFC5482] L. Eggert. *TCP User Timeout Option*. RFC5482. Mar. 2009. URL: https://tools.ietf.org/html/rfc5482.

[RFC2883] S. Floyd. *An Extension to the Selective Acknowledgement (SACK) Option for TCP*. RFC2883. July 2000. URL: https://tools.ietf.org/html/rfc2883.

[RFC3649] S. Floyd. *Highspeed TCP for Large Congestion Windows*. RFC3649. Dec. 2003. URL: https://tools.ietf.org/html/rfc3649.

[6] GNU. *Gawk*. URL: https://www.gnu.org/software/gawk/gawk.html.

[7] GNU. *GNU Bash*. URL: https://www.gnu.org/software/bash/bash.html.

[8] GNU. *sed*. URL: https://www.gnu.org/software/sed/.

[RFC6093] F. Gont. *On the Implementation of the TCP Urgent Mechanism*. RFC6093. 1. URL: https://tools.ietf.org/html/rfc6093.

[9] Sangtae Ha, Injong Rhee, and Lisong Xu. "CUBIC: A New TCP-friendly High-speed TCP Variant". In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 64–74. ISSN: 0163-5980. DOI: 10.1145/1400097.1400105. URL: http://doi.acm.org/10.1145/1400097.1400105.

[RFC2861] M. Handley. *TCP Congestion Window Validation*. RFC2861. July 2000. URL: https://www.ietf.org/rfc/rfc2861.txt.

[RFC6582] T. Henderson. *The NewReno Modification to TCP's Fast Recovery Algorithm*. RFC6582. 4. URL: https://tools.ietf.org/html/rfc6582.

[10] Tom Herbert. *bql: Byte Queue Limits*. 2011. URL: https://lwn.net/Articles/469652/.

[11] Tomas Hruby. *Byte Queue Limits*. 2012. URL: http://www.linuxplumbersconf.org/2012/wp-content/uploads/2012/08/bql_slide.pdf.

[12] Iperf. *Iperf*. URL: https://iperf.fr/.

[RFC1144] V. Jacobson. *Compressing TCP/IP Headers for Low-Speed Serial Links*. RFC1144. Feb. 1990. URL: https://tools.ietf.org/html/rfc1144.

[RFC1323] V. Jacobson. *TCP Extensions for High Performance*. RFC1323. May 1992. URL: https://www.ietf.org/rfc/rfc1323.txt.

[13] Tom Kelly. "Scalable TCP: Improving Performance in Highspeed Wide Area Networks". In: *SIGCOMM Comput. Commun. Rev.* 33.2 (Apr. 2003), pp. 83–91. ISSN: 0146-4833. DOI: 10.1145/956981.956989. URL: http://doi.acm.org/10.1145/956981.956989.

[RFC5562] A. Kuzmanovic. *Adding Explicit Congestion Notification (ECN) Capability to TCP's SYN/ACK Packets*. RFC5562. June 2009. URL: https://tools.ietf.org/html/rfc5562.

[14] Mathieu Lacage. *DCE (Direct Code Execution)*. URL: https://www.nsnam.org/overview/projects/direct-code-execution/ (visited on 03/23/2015).

[drafthtcp] D. Leith. *H-TCP: TCP Congestion Control for High Bandwidth-Delay Product Paths*. drafthtcp. June 2007. URL: https://tools.ietf.org/html/draft-leith-tcp-htcp-03.

[RFC3522] R. Ludwig. *The Eifel Detection Algorithm for TCP*. RFC3522. Apr. 2003. URL: https://tools.ietf.org/html/rfc3522.

[15] J. Mahdavi M. Mathis. "Forward acknowledgement: refining TCP congestion control". In: *Computer Communication Review*. SIGCOMM '96 Conference proceedings on Applications, technologies, architectures, and protocols for computer communications. New York: ACM, 1996, pp. 281–291.

[draftprr] Y. Cheng M. Mathis N. Dukkipati. *Proportional Rate Reduction for TCP*. draftprr. July 2011. URL: https://tools.ietf.org/html/draft-mathis-tcpm-proportional-rate-reduction-01.

[RFC2018] J. Mahdavi. *TCP Selective Acknowledgment Options*. RFC2018. Oct. 1996. URL: https://tools.ietf.org/html/rfc2018.

[RFC4821] M. Mathis. *Packetization Layer Path MTU Discovery*. RFC4821. Mar. 2007. URL: https://www.ietf.org/rfc/rfc4821.txt.

[RFC6937] M. Mathis. *Proportional Rate Reduction for TCP*. RFC6937. May 2013. URL: https://tools.ietf.org/html/rfc6937.

[RFC1191] J. Mogul. *Path MTU Discovery*. RFC1191. Nov. 1990. URL: https://tools.ietf.org/html/rfc1191.

[16] Salemn Nasri, Mohammed Othman, and Rachid Mbarek. "Fairness of High-Speed TCP Protocols with Different Flow Capacities". In: *Journal Of Networks* 4.3 (2009), pp. 163–169. ISSN: 1796-2056. DOI: 10.4304/jnw.4.3.163-169.

[17] Netperf. *Netperf*. URL: http://www.netperf.org/netperf/.

[draftpie] R. Pan. *PIE: A Lightweight Control Scheme To Address the Bufferbloat Problem*. draftpie. Dec. 2012. URL: https://tools.ietf.org/html/draft-pan-tsvwg-pie-00.

[RFC3168] K. Ramakrishnan. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC3168. July 2001. URL: https://tools.ietf.org/html/rfc3168.

[RFC5682] P. Sarolahti. *Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP*. RFC5682. Sept. 2009. URL: https://tools.ietf.org/html/rfc5682.

[RFC6013] W. Simpson. Jan. 2011.

[RFC793] Information Sciences Institute University of Southern California. *TRANSMISSION CONTROL PROTOCOL*. RFC7414. Sept. 1981. URL: https://tools.ietf.org/html/rfc793.

[RFC2140] J. Touch. *TCP Control Block Interdependence*. RFC2140. Apr. 1997. URL: https://tools.ietf.org/html/rfc1144.

[18] Bake Tutorial. *Bake Tutorial*. URL: https://www.nsnam.org/docs/bake/tutorial/html/bake-over.html.

[19] Waf. *The Waf Build System*. URL: https://waf.io.

[20] Li Yee Ting, D. Leith, and R.N Shorten. "Experimental Evaluation of TCP Protocols for High-Speed Networks". In: *Networking, IEEE/ACM Transactions* 15 (5 2007), pp. 1109–112. ISSN: 1063-6692. DOI: 10.1109/TNET.2007.896240.